O'REILLY xml.com
xml from the inside out

PORTAL WORLD.ws
Opening the Windows to your Portal Solutions

JBuilder 4 Handheld Express
Product Review
Find it at ONJava.com

Add the latest news from XML.com to your web site
Click here!

What is XSLT?

by G. Ken Holman
August 16, 2000

Introduction

Now that we are successfully using XML to mark up our information according to our own vocabularies, we are taking control and responsibility for our information, instead of abdicating such control to product vendors. These vendors would rather lock our information into their proprietary schemes to keep us beholden to their solutions and technology.

IBM senior programmer Doug Tidwell will be speaking on Java Techniques for XSLT Web Sites at the O'Reilly Conference on Enterprise Java, March 26-29, in Santa Clara, California.

But the flexibility inherent in the power given to each of us to develop our own vocabularies, and for industry associations, e-commerce consortia, and the W3C to develop their own vocabularies, presents the need to be able to transform information marked up in XML from one vocabulary to another.

Two W3C Recommendations, XSLT (the Extensible Stylesheet Language Transformations) and XPath (the XML Path Language), meet that need. They provide a powerful implementation of a tree-oriented transformation language for transmuting instances of XML using one vocabulary into either simple text, the legacy HTML vocabulary, or XML instances using any other vocabulary imaginable. We use the XSLT language, which itself uses XPath, to specify how an implementation of an XSLT processor is to create our desired output from our given marked-up input.

Sponsored By:

Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

XSLT enables and empowers interoperability. This XML.com introduction strives to overview essential aspects of understanding the context in which these languages help us meet our transformation requirements, and to introduce substantive concepts and terminology to bolster the information available in the W3C Recommendation documents themselves.

Since April 1999 Crane Softwrights Ltd. has published commercial training material titled Practical Transformation Using XSLT and XPath, covering the entire scope of the W3C XSLT and XPath through working drafts and the final 1.0 recommendations. This material is delivered by Crane in instructor-led sessions and is licensed to other training organizations around the world needing to teach these exciting technologies.

Crane has rewritten the first two chapters of this material into prose. These prose-oriented chapters are published on XML.com correspondingly as two main sections. The material assumes no prior knowledge of XSLT and XPath and guides the reader through background, context, structure, concepts and introductory terminology.

Table of Contents

1. The context of XSL Transformations and the XML Path Language
- 1.1 The XML family of Recommendations
  - 1.1.1 Extensible Markup Language (XML)
  - 1.1.2 XML Path Language (XPath)
  - 1.1.3 Styling structured information
  - 1.1.4 Extensible Stylesheet Language (XSL)
  - 1.1.5 Extensible StylesheetLanguage Transformations (XSLT)
  - 1.1.6 Namespaces
  - 1.1.7 Stylesheet association
- 1.2 Transformation data flows
  - 1.2.1 Transformation from XML to XML
  - 1.2.2 Transformation from XML to XSL formatting semantics
  - 1.2.3 Transformation from XML to non-XML
  - 1.2.4 Three-tiered architectures

2. Getting started with XSLT and XPath
- 2.1 Stylesheet examples
  - 2.1.1 Some simple examples
  - 2.1.2 Some more complex examples
- 2.2 Syntax basics - stylesheets, templates, instructions
  - 2.2.1 Explicitly declared

Sponsored By:

stylesheets
·2.2.2 Implicitly declared stylesheets
·2.2.3 Stylesheet requirements
·2.2.4 Instructions and literal result elements
·2.2.5 Templates and template rules
·2.2.6 Approaches to stylesheet design

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

Crane Softwrights Training Information

To follow this link, please proceed to http://www.CraneSoftwrights.com/training/.

If your browser supports automatic redirection, this link will be automatically traversed.

Otherwise, please click on the link above to proceed!

Crane Softwrights Ltd.

Free download previews

Practical Transformation Using XSLT and XPath (free 137-page download preview in 2-up pages in PDF; 1-up and 2-up are available for purchase; published review):

- US-letter size paper - 594,809 bytes zipped - **free download**
- A4 size paper - 596,446 bytes zipped - **free download**

Purchasing information links

- *** Click here for on-line pricing and purchasing information, including individual, site-wide staff (all staff members in a given location) licenses or world-wide staff (all staff members world-wide) licenses, all with perpetual free updates.***
- Note that we will also directly accept faxed-in purchase orders on company letterhead and mailed-in cheques for the amounts described in the above link.

Details follow below. We would appreciate any feedback you have, or suggestions for changes and improvements; please forward your comments to training@CraneSoftwrights.com.

CRANE SOFTWRIGHTS LTD.

**BOX 266,
KARS, ONTARIO
CANADA K0A-2E0**

**+1 (613) 489-0999 (Voice)
+1 (613) 489-0995 (Fax)**

1. Purchasable Materials
2. Staff Licenses
3. On-line CBT
4. Printed books by others

Training Programmes and Training Materials

When face-to-face training is not an option for you (see http://www.CraneSoftwrights.com/schedule.htm for details), we hope you will find these training programmes and materials of use.

This page also describes the general information regarding our free downloads of overview and helpful reference material, the policy of *free access to revisions of printed materials* for registered purchasers, and a description of the staff licenses available in addition to individual purchases.

1. Course/Presentation Materials

Course materials are structured as tutorial references with detailed examples and quick references to use as supplemental material to published standards and recommendations.

Summary of Overviews Available for Free Download:

- Practical Transformation Using XSLT and XPath
  XSL Transformations and the XML Path Language
  Ninth Edition - ISBN 1-894049-06-3 - 2001-01-19

Why be interested in purchasing a complete set of materials? A printed book is often out of date soon after hitting the streets. Crane's training materials used during face-to-face training sessions are kept up to date as the information being taught changes. Our policy of offering free updates to purchasers of our training materials ensures you have the latest version of our tutorial information. In effect, our publications are edited by our customers in that suggestions for clarifications, improvements or enhanced examples are considered for inclusion in future editions. The publications are *only* made available electronically in Adobe PDF (click here to obtain the Adobe Acrobat PDF Reader for free; users who are obliged to use GhostScript must note the files utilize features of PostScript 3 that are *not* supported in GhostScript 5.5, but are anticipated to be available in GhostScript 6.0 when released). The information can be obtained in full size or compact presentation forms.

With the purchase of a registered copy of the materials, ***you are entitled to request copies of updates to the material at no extra cost after every time they are revised***, thus your reference is always the latest version published by Crane. The publishing plan for each publication is noted below in each description.

Important Note: Your purchase of the materials is for your own use only. The password access to the materials entitles you to download the PDF file for you to print or use without sharing with others. Please have others obtain their own copy of these publications for their use. *Thank you!*

Purchasers have ten different publishing formats to choose from:

- A4 - full page - single sided (????-a4.pdf) - bound
- A4 - full page - double sided (????-a4-dbl.pdf) - long edge duplex; bound
- A4 - full page - 2-up per page (????-a4-2up.pdf) - optional long or short edge duplex
- A4 - half page - single sided (????-a4-bind.pdf) - cut, stacked, bound
- A4 - half page - double sided (????-a4-bind-dbl.pdf) - short edge duplex, cut, stacked, bound
- US letter - full page - single sided (????-us.pdf) - bound
- US letter - full page - double sided (????-us-dbl.pdf) - long edge duplex; bound
- US letter - full page - 2-up per page (????-us-2up.pdf) - optional long or short edge duplex
- US letter - half page - single sided (????-us-bind.pdf) - cut, stacked, bound
- US letter - half page - double sided (????-us-bind-dbl.pdf) - short edge duplex, cut, stacked, bound

Notes:

- "bound" versions have their margin adjusted for left edge hole punching or binding

- "cut, stacked" refers to the act of cutting the pages in half after being printed and stacking the left stack on top of the right stack before binding
- "short edge duplex" and "long edge duplex" refer to the orientation edge when printing double sided
- a separate ZIP file with all XML and XSLT files used in the material can be downloaded separately

See the Crane Course Schedule for more details of when these published materials are delivered face-to-face at conferences or host locations.

For each of the course or presentation publications available, the overview pages from each module are collected above for free download and distribution to review the content of the publication.

To be informed of the availability of this material for purchase, please send your request to publications@CraneSoftwrights.com

1.1 Introduction to XSLT

Third Edition - ISBN 1-894049-00-4 - 1999-06-08

This publication has been entirely replaced by Practical Transformation Using XSLT and XPath and is no longer available. All customers of this (and other editions) have equal access to any replacement publication.

1.2 Practical Transformation Using XSLT and XPath

XSL Transformations and the XML Path Language
Ninth Edition - ISBN 1-894049-06-3 - 2001-01-19

This comprehensive guide to XSL Transformations (XSLT) and the XML Path Language (XPath) according to the XSLT/XPath 19991116 1.0 Recommendations is over 300 pages of explanatory material, diagrams, tables, and code samples. Every markup construct used for XSLT and XPath is identified and described. The focus is primarily on the W3C work and not on archaic definitions or implementations.

Important note: There are copies of a prose re-write of a two-chapter excerpt of the eighth edition posted publicly on the web, though the ***purchasable product is not in prose, rather, it is in a detailed bulleted format*** (all that is missing is the sentence structure, not any content). Please review the free download excerpt to see the exact nature of the materials as they are currently available for purchase, as the purchase is non-refundable. All future editions will be freely available to all registered customers of the current work. The nature of future work is now focused on keeping the material up-to-date, not on the re-writing of the content into prose. The W3C XSL Working Group has announced their new charter at http://www.w3.org/Style/2000/xsl-charter.html indicating upcoming revisions to the Recommendations. We plan to keep this material up-to-date with revisions to the Recomendations and to continue to re-issue new editions of your purchase as the Recommendations change or as we receive sufficient feedback to warrant releasing new material.

Free Download at top of this page: Module Introductions and Preview only - 2001-01-19 - over 130 pages including the complete text of the first two and last two modules is included to illustrate the bulleted nature of the content and the level of detail of the remainder of the materials; the last two modules include cross reference information enhanced from the W3C documents as well as

illustrative documentation for XT and Microsoft IE5.

Pricing information is available at the top of this page: note that the purchase includes subscription to all future updates of the same material whether you buy an individual copy, a copy for a single site's local intranet, or a copy for a world-wide corporate intranet.

There is a published review of the book and a brief public testimonial regarding this work on the XSL List. We encourage all suggestions to improve the materials in order that existing customers can get updated publications and future customers get as complete a collection of information as possible.

2. Staff Licenses

The same purchase rights given to an individual for perpetual no-charge updates to a purchased book is granted to staff members in either a site-wide staff license or a world-wide staff license.

Your organization can give your staff, but not your customers, access to a copy of a Crane book on your own intranet provided it is protected from outside access.

The site-wide staff license is granted to all staff members whose office is at a single physical mailing address.

The world-wide staff license is granted to all staff members world-wide of the company making the purchase.

In each case this is a one-time fee, with perpetual free access to updates.

Pricing information is available at the top of this page

3. On-line Web-based Computer Based Training (CBT)

Due to a lack of sufficient interest and to supplier problems, we are no longer actively pursuing web-based training providers (we had hoped for the resources to host our courses in a web-based forum for interactive training with self-assessment).

To express your interest or to be informed of any possible future availability of these courses, please send your request to online-cbt@CraneSoftwrights.com.

4. Printed books by others

Electronic publications are not for everyone. Even with a ZIP file of samples, free updates, and the ability to search the PDF for information in the book, some people are just not oriented to reading electronic books.

While some people do take our electronic materials and bind their own printed copies, others are only comfortable with a physical book in their hands. We have begun a list of related books written by others as a service to our visitors. The list is not meant to prejudice other books that have not made it on the list: if you know of a title we should consider adding to the list, please let us know.

More Information

For more information please see our home page at: http://www.CraneSoftwrights.com or email us at training@CraneSoftwrights.com.

$Date: 2001/04/01 00:23:51 $(UTC)

O'REILLY xml.com
xml from the inside out

XML europe 2001 CONFERENCE & EXPOSITION
21-25 May 2001 • Berlin, Germany www.xmleurope.com

This Article

What is XSLT? (I)

by G. Ken Holman
August 16, 2000

The Context of XSL Transformations and the XML Path Language

This first chapter examines the context of two W3C Recommendations -- Extensible Stylesheet Language Transformations (XSLT) and XML Path Language (XPath) -- within the growing family of Recommendations related to the Extensible Markup Language (XML). Later we will look at detailed examples, but first let's focus on XSLT and XPath in the context of a few of the Recommendations in the XML family and examine how these two Recommendations work together to address separate and distinct functionality required when working with structured information technologies.

This chapter does not attempt to address all of the numerous XML-related Recommendations currently released or in development. Specifically, we will be looking at only the following as they relate to XSLT and XPath:

Extensible Markup Language (XML)

For years, applications and

Table of Contents

1. The context of XSL Transformations and the XML Path Language
- 1.1 The XML family of Recommendations
  - 1.1.1 Extensible Markup Language (XML)
  - 1.1.2 XML Path Language (XPath)
  - 1.1.3 Styling structured information
  - 1.1.4 Extensible Stylesheet Language (XSL)
  - 1.1.5 Extensible Stylesheet Language Transformations (XSLT)
  - 1.1.6 Namespaces
  - 1.1.7 Stylesheet association
- 1.2 Transformation data flows
  - 1.2.1 Transformation from XML to XML
  - 1.2.2 Transformation from XML to XSL formatting semantics
  - 1.2.3 Transformation from XML to non-XML

Sponsored By:

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

·1.2.4 Three-tiered architectures

vendors have imposed their constraints on the way we can represent our information. Our data has been created, maintained, stored and archived according to the rules enforced by others. The advent of the Extensible Markup Language (XML) moves the control of our information out of the hands of others and into our *own* by providing two basic facilities.

XML describes rules for structuring our information using embedded markup of our own choice. We can take control of our information representation by creating and using a vocabulary we design of elements and attributes that makes sense for the way we do our business and use our data.

In addition, XML describes a language for formally declaring the vocabularies we use. This allows our tools to constrain the creation of an instance of our information, and allows our users to validate a properly created instance of information against our set of constraints.

Note 1:	An XML *document* is just an instance of well-formed XML. The two terms *document* and *instance* could be used interchangeably, but this reference material uses the term *instance* to help readers remember that XML isn't just for documents or documentation. With XML we describe a related set of information in a tree-like hierarchical fashion, and gain the benefits of having done so, whether the information captures an invoice-related transaction between computers, or the content of a user manual rendered on paper.

Sponsored By:

XML Path Language (XPath)

XPath is a string syntax for building addresses to the information found in an XML document. We use this language to specify the locations of document structures or data found in an XML document when processing that information using XSLT. XPath allows us from any location to address any other location or content.

Extensible Stylesheet Language Family (XSLT/XSL)

Two vocabularies specified in separate W3C Recommendations provide for the two distinct styling processes of transforming and rendering XML instances.

We can transform information using one vocabulary into an alternate form by using the Extensible Stylesheet Language Transformations (XSLT).

The Extensible Stylesheet Language (XSL) is a rendering vocabulary describing the semantics of formatting information for different media.

Namespaces

We use XML namespaces to distinguish information when mixing multiple vocabularies in a single instance. Without namespaces our processes would find the information ambiguous when identical names have been chosen by the designers of the vocabularies we use.

Stylesheet Association

We declare our choice of an associated stylesheet for an XML instance by embedding the construct described in the Stylesheet Association Recommendation. Recipients and applications can choose to respect or ignore this choice, but the declaration indicates that we have tied some

process (typically rendering) to our data, which specifies how to consume or work with our information.

1.1 The XML family of Recommendations

Now let's look at the objectives of these selected Recommendations.

1.1.1 Extensible Markup Language (XML)

Historically, the ways we have expressed, created, stored and transmitted our electronic information have been constrained and controlled by the vendors we choose and the applications we run. Alternatively, we now can express our data in a structured fashion oriented around our perspective of the nature of the information itself rather than the nature of an application's choice of how to represent our information. With Extensible Markup Language (XML), we describe our information using embedded markup of elements, attributes and other constructs in a tree-like structure.

- http://www.w3.org/TR/REC-xml

1.1.1.1 Structuring information

Contrasted to a file format where information identification relies on some proprietary hidden format, predetermined ordering, or some kind of explicit labeling, the tree-like hierarchical storage structure infers relationships by the scope of values encompassing the scopes of other values.

Though trees shape a number of areas of XML, both logically (markup) and physically (entities such as files or other resources), they are not the only means by which relationships are specified. For example, a quantum of information can arbitrarily point or refer to other information elsewhere through use of unique identifiers.

Two basic objectives of representing information hierarchically are satisfied by the XML Recommendation. It provides:

- an unambiguous mechanism for constraining structure in a stream of information

  XML defines the concept of well-formedness. Well-formedness dictates the syntax used for markup languages within the content of an instance of information. This is the syntax of using angle brackets ("<" and ">") and the ampersand ("&") to demarcate and identify constituent components of information within a file, a resource or a bound data stream. Users of the Hypertext Markup Language (HTML) will recognize the use of these characters for marking the vocabulary described by the designers of the World Wide Web in their web documents.

- a language for specifying how a system can constrain the allowed logical hierarchy of information structures

  XML defines the concept of validity with a syntax for a meta-markup language used to specify vocabularies. A Document Type Definition (DTD) describes the structural schema mandating the user-defined constraints on well-formed information. The designers of HTML have formalized their vocabulary through such a DTD, thus declaring the allowed or expected relationships between components of a hypertext document.

There is an implicit document model for an instance of well-formed

XML defined by the mere presence of nested elements found in the information. There is no need to declare this model because the syntax rules governing well-formedness guarantee the information to be seen properly as a hierarchy. As with all hierarchies, there are family-tree-like relationships of parent, child, and sibling constructs relative to each construct found.

Consider the following well-formed XML instance `purc.xml`:

```
01  <?xml version="1.0"?>
02  <purchase id="p001">
03    <customer db="cust123"/>
04    <product db="prod345">
05      <amount>23.45</amount>
06    </product>
07  </purchase>
```

Example 1-1: A well-formed XML purchase order instance.

Observe the content nesting (whitespace has been added only for illustrative purposes). The instance follows the lexical rules for XML markup and the hierarchical model is implicit by the nesting of elements. Pay particular attention to the markup on line 3 for the empty element named `customer`, with the attribute named `db`. It will be used later in examples throughout this chapter. The customer element is a child of the document element, which is named `purchase`.

Although the presence of an explicit formal document model is useful to an XML processor or to a system working with XML instances, that model has no impact on the implicit structural model and only minor influence on the interpretation of content found in the instance. This point holds true whether the model is expressed in a DTD or in some of the other Recommendations for structural and content schemata being developed.

Consider the following valid XML instance `purcdtd.xml`:

```
01  <?xml version="1.0"?>
02  <!DOCTYPE purchase [
03  <!ELEMENT purchase ( customer, product+ )>
04  <!ATTLIST purchase id ID #REQUIRED>
05  <!ELEMENT customer EMPTY>
06  <!ATTLIST customer db CDATA #REQUIRED>
07  <!ELEMENT product  ( amount )>
08  <!ATTLIST product  db CDATA #REQUIRED>
09  <!ELEMENT amount   ( #PCDATA )>
10  ]>
11  <purchase id="p001">
12    <customer db="cust123"/>
13    <product db="prod345">
14      <amount>23.45</amount>
15    </product>
16  </purchase>
```

Example 1-2: A valid XML purchase order instance

See how the information content is no different from the previous example, but in this case an explicit document model using XML 1.0 DTD syntax is included (it could have been included by reference to a separate resource). A processor can validate that the information content conforms not only to the lexical rules for XML (well-formedness) but

also the syntax rules dictated by the supplied document model (validity).

Looking at the same `customer` element as before (now on line 12), the document model indicates on line 6 that the `db` attribute is, indeed, required: if the attribute is absent the XML processor can report syntactic model constraint violation even if the element is otherwise lexically well-formed. The document model can also provide additional information not evident without a document model (such as the information on line 4 that the `id` attribute for `purchase` is of XML type `ID`).

1.1.1.2 No built-in meanings or concepts

The area of semantics associated with XML instances is very gray. A document model is but one component used to help describe the semantics of the information found in an instance. While well-formed instances do not have a formal document model, often the names of the constructs used within the instances give hints to the associated semantics. Without a formalism yet available in our community to express semantics in a rigorous fashion, we users of XML do (or should!) capture the semantics of a given vocabulary in prose, whether or not the document model is formalized.

The XML 1.0 Recommendation only describes the behavior required of an XML processor acting on an XML stream, and how it must identify constituent data and provide that data to an application using the processor:

Since there are no formalized semantic description facilities in XML, any XML that is used is not tied to any one particular concept or application. There are no rendition or transformation rules or constructs defined in XML. The only purpose of XML is to unambiguously identify and deliver constituent components of data. There are no inherent meanings or semantics of any kind associated with element types defined in a document model. There are no defined controls for implying any rendering semantics.

Even the `xml:space` attribute allowing for the differentiation of whitespace found in a document is not an aspect of rendering but of information description. The author or modeler of an instance is indicating with this reserved attribute (termed "special" in XML 1.0) the nature of the information and how the whitespace found in the information is to be either preserved or handled by a processor in a default fashion.

Some new users of XML who have a background in a markup language such as HTML often assume a magical association of semantics with element types of the same names they have been exposed to in their prior work. In a web page, they can safely assume that the construct `<p>` will be interpreted as a paragraph or `<em>` as emphasized text. However, this interpretation is solely the purview of the designers of HTML and user agents attempting to conform to the World Wide Web Consortium (W3C)-published semantics. Nothing is imposed by any process when creating a new XML vocabulary that happens to use the same names. Applications using XML processors to access XML information must be instructed how to interpret and implement the desired semantics.

1.1.2 XML Path Language (XPath)

Assuming that we have structured our information using XML, how are we going to talk about (address) what is inside our documents? Locating

information in an XML document is critical to both transforming it and to associating or relating it to other information. When we write stylesheets and use linking languages, we can address components of our information for a processor by our use of the XML Path Language, also called XPath:

- http://www.w3.org/TR/xpath

1.1.2.1 Addressing structured information

The W3C working group responsible for stylesheets collaborated with the W3C working group responsible for the next generation of hyperlinking to produce XPath as a common base for addressing requirements shared by their respective Recommendations. Both groups extend the core XPath facilities to meet the needs they have in each of their domains: the stylesheet group uses XPath as the core of expressions in XSLT; the linking group uses XPath as the core of expressions in the XPointer Recommendation.

In order to address components you have to know the addressing scheme with which the components are arranged. The basis of addressing XML documents is an abstract data model of interlinked nodes arranged hierarchically echoing the tree-shape of the nested elements in an instance. Nodes of different types make up this hierarchy, each node representing the parsed result of a syntactic structure found in the bytes of the XML instance.

This abstraction insulates addressing from the multiple syntactic forms of given XML constructs, allowing us to focus on the information itself and not the syntax used to represent the information.

Note 2:	We see XML documents as a stream or string of bytes that follow the rules of the XML 1.0 Recommendation. Stylesheets do not regard instances in this fashion, and we have to change the way we think of our XML documents in order to successfully work with our information. This leap of understanding ranks high on the list of key aspects of stylesheet writing I needed to internalize before successfully using this technology.

We are given tools to work in the framework provided by the abstraction: a set of data types used to represent values found in the generalization, and a set of functions we use to manipulate and examine those values. The data types include strings, numbers, boolean values and sets of nodes of our information. The functions allow us to cast these values into other data type representations and to return massaged information according to our needs.

1.1.2.2 Addressing identifies a hierarchical position or positions

XPath defines common semantics and syntax for addressing XML-expressed information, and bases these primarily on the hierarchical position of components in the tree. This ordering is referred to as document order in XPath, while in other contexts this is often termed either parse order or depth-first order. Alternatively, we can access an arbitrary location in the tree based on points in the tree having unique identifiers.

We convey XPath addresses in a simple and compact non-XML syntax. This allows us to use an XPath expression as the value of an attribute in an XML vocabulary as in the following examples:

```
01  select="answer"
```

Example 1-3: A simple XPath expression in a `select` attribute

The above attribute value expresses all children named "`answer`" of the current focus element.

```
01  match="question|answer"
```

Example 1-4: An XPath expression in a `match` attribute

The above attribute value expresses a test of an element being in the union of the element types named "`question`" and "`answer`".

The XPath syntax looks a lot like addressing subdirectories in a file system or as part of a Universal Resource Identifier (URI). Multiple steps in a location path are separated by either one or two oblique "`/`" characters. Filters can be specified to further refine the nature of the components of our information being addressed.

```
01  select="question[3]/answer[1]"
```

Example 1-5: A multiple step XPath expression in a `select` attribute

The above example selects only the first "`answer`" child of the third "`question`" child of the focus element.

```
01  select="id('start')//question[@answer='y']"
```

Example 1-6: A more complex XPath expression in a `select` attribute

The above example uses an XPath address identifying some descendants of the element in the instance that has the unique identifier with the value "`start`". Those identified are the question elements whose answer attribute is equal to the string equal to the lower-case letter '`y`'. The value returned is the set of nodes representing the elements meeting the conditions expressed by the address. The address is used in a `select` attribute, thus the XSLT processor is selecting all of the addressed elements for some kind of processing.

1.1.2.3 XPath is *not* a query language

It is important to remember that addressing information is only one aspect of querying information. Other aspects include query operators that massage intermediate results into a final result. While a few operators and functions are available in XSLT to use values identified in documents, these are oriented to string processing, not to complex operations required by some applications.

Note 3:	When query Recommendations are developed, I would hope that the addressing portion is based on XPath as a core, just as with XSLT.

Pages: **1**, 2, 3

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

This Article

What is XSLT? (I)

by G. Ken Holman
August 16, 2000

The Context of XSL Transformations and the XML Path Language

This first chapter examines the context of two W3C Recommendations -- Extensible Stylesheet Language Transformations (XSLT) and XML Path Language (XPath) -- within the growing family of Recommendations related to the Extensible Markup Language (XML). Later we will look at detailed examples, but first let's focus on XSLT and XPath in the context of a few of the Recommendations in the XML family and examine how these two Recommendations work together to address separate and distinct functionality required when working with structured information technologies.

This chapter does not attempt to address all of the numerous XML-related Recommendations currently released or in development. Specifically, we will be looking at only the following as they relate to XSLT and XPath:

Extensible Markup Language (XML)

For years, applications and

Table of Contents

1. The context of XSL Transformations and the XML Path Language
- 1.1 The XML family of Recommendations
  - 1.1.1 Extensible Markup Language (XML)
  - 1.1.2 XML Path Language (XPath)
  - 1.1.3 Styling structured information
  - 1.1.4 Extensible Stylesheet Language (XSL)
  - 1.1.5 Extensible Stylesheet Language Transformations (XSLT)
  - 1.1.6 Namespaces
  - 1.1.7 Stylesheet association
- 1.2 Transformation data flows
  - 1.2.1 Transformation from XML to XML
  - 1.2.2 Transformation from XML to XSL formatting semantics
  - 1.2.3 Transformation from XML to non-XML

Sponsored By:

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

·1.2.4 Three-tiered architectures

vendors have imposed their constraints on the way we can represent our information. Our data has been created, maintained, stored and archived according to the rules enforced by others. The advent of the Extensible Markup Language (XML) moves the control of our information out of the hands of others and into our *own* by providing two basic facilities.

XML describes rules for structuring our information using embedded markup of our own choice. We can take control of our information representation by creating and using a vocabulary we design of elements and attributes that makes sense for the way we do our business and use our data.

In addition, XML describes a language for formally declaring the vocabularies we use. This allows our tools to constrain the creation of an instance of our information, and allows our users to validate a properly created instance of information against our set of constraints.

Note 1:	An XML *document* is just an instance of well-formed XML. The two terms *document* and *instance* could be used interchangeably, but this reference material uses the term *instance* to help readers remember that XML isn't just for documents or documentation. With XML we describe a related set of information in a tree-like hierarchical fashion, and gain the benefits of having done so, whether the information captures an invoice-related transaction between computers, or the content of a user manual rendered on paper.

Sponsored By:

XML Path Language (XPath)

XPath is a string syntax for building addresses to the information found in an XML document. We use this language to specify the locations of document structures or data found in an XML document when processing that information using XSLT. XPath allows us from any location to address any other location or content.

Extensible Stylesheet Language Family (XSLT/XSL)

Two vocabularies specified in separate W3C Recommendations provide for the two distinct styling processes of transforming and rendering XML instances.

We can transform information using one vocabulary into an alternate form by using the Extensible Stylesheet Language Transformations (XSLT).

The Extensible Stylesheet Language (XSL) is a rendering vocabulary describing the semantics of formatting information for different media.

Namespaces

We use XML namespaces to distinguish information when mixing multiple vocabularies in a single instance. Without namespaces our processes would find the information ambiguous when identical names have been chosen by the designers of the vocabularies we use.

Stylesheet Association

We declare our choice of an associated stylesheet for an XML instance by embedding the construct described in the Stylesheet Association Recommendation. Recipients and applications can choose to respect or ignore this choice, but the declaration indicates that we have tied some

process (typically rendering) to our data, which specifies how to consume or work with our information.

1.1 The XML family of Recommendations

Now let's look at the objectives of these selected Recommendations.

1.1.1 Extensible Markup Language (XML)

Historically, the ways we have expressed, created, stored and transmitted our electronic information have been constrained and controlled by the vendors we choose and the applications we run. Alternatively, we now can express our data in a structured fashion oriented around our perspective of the nature of the information itself rather than the nature of an application's choice of how to represent our information. With Extensible Markup Language (XML), we describe our information using embedded markup of elements, attributes and other constructs in a tree-like structure.

- http://www.w3.org/TR/REC-xml

1.1.1.1 Structuring information

Contrasted to a file format where information identification relies on some proprietary hidden format, predetermined ordering, or some kind of explicit labeling, the tree-like hierarchical storage structure infers relationships by the scope of values encompassing the scopes of other values.

Though trees shape a number of areas of XML, both logically (markup) and physically (entities such as files or other resources), they are not the only means by which relationships are specified. For example, a quantum of information can arbitrarily point or refer to other information elsewhere through use of unique identifiers.

Two basic objectives of representing information hierarchically are satisfied by the XML Recommendation. It provides:

- an unambiguous mechanism for constraining structure in a stream of information

  XML defines the concept of well-formedness. Well-formedness dictates the syntax used for markup languages within the content of an instance of information. This is the syntax of using angle brackets ("<" and ">") and the ampersand ("&") to demarcate and identify constituent components of information within a file, a resource or a bound data stream. Users of the Hypertext Markup Language (HTML) will recognize the use of these characters for marking the vocabulary described by the designers of the World Wide Web in their web documents.

- a language for specifying how a system can constrain the allowed logical hierarchy of information structures

  XML defines the concept of validity with a syntax for a meta-markup language used to specify vocabularies. A Document Type Definition (DTD) describes the structural schema mandating the user-defined constraints on well-formed information. The designers of HTML have formalized their vocabulary through such a DTD, thus declaring the allowed or expected relationships between components of a hypertext document.

There is an implicit document model for an instance of well-formed

XML defined by the mere presence of nested elements found in the information. There is no need to declare this model because the syntax rules governing well-formedness guarantee the information to be seen properly as a hierarchy. As with all hierarchies, there are family-tree-like relationships of parent, child, and sibling constructs relative to each construct found.

Consider the following well-formed XML instance `purc.xml`:

```
01  <?xml version="1.0"?>
02  <purchase id="p001">
03    <customer db="cust123"/>
04    <product db="prod345">
05      <amount>23.45</amount>
06    </product>
07  </purchase>
```

Example 1-1: A well-formed XML purchase order instance.

Observe the content nesting (whitespace has been added only for illustrative purposes). The instance follows the lexical rules for XML markup and the hierarchical model is implicit by the nesting of elements. Pay particular attention to the markup on line 3 for the empty element named `customer`, with the attribute named `db`. It will be used later in examples throughout this chapter. The customer element is a child of the document element, which is named `purchase`.

Although the presence of an explicit formal document model is useful to an XML processor or to a system working with XML instances, that model has no impact on the implicit structural model and only minor influence on the interpretation of content found in the instance. This point holds true whether the model is expressed in a DTD or in some of the other Recommendations for structural and content schemata being developed.

Consider the following valid XML instance `purcdtd.xml`:

```
01  <?xml version="1.0"?>
02  <!DOCTYPE purchase [
03  <!ELEMENT purchase ( customer, product+ )>
04  <!ATTLIST purchase id ID #REQUIRED>
05  <!ELEMENT customer EMPTY>
06  <!ATTLIST customer db CDATA #REQUIRED>
07  <!ELEMENT product  ( amount )>
08  <!ATTLIST product  db CDATA #REQUIRED>
09  <!ELEMENT amount   ( #PCDATA )>
10  ]>
11  <purchase id="p001">
12    <customer db="cust123"/>
13    <product db="prod345">
14      <amount>23.45</amount>
15    </product>
16  </purchase>
```

Example 1-2: A valid XML purchase order instance

See how the information content is no different from the previous example, but in this case an explicit document model using XML 1.0 DTD syntax is included (it could have been included by reference to a separate resource). A processor can validate that the information content conforms not only to the lexical rules for XML (well-formedness) but

also the syntax rules dictated by the supplied document model (validity).

Looking at the same `customer` element as before (now on line 12), the document model indicates on line 6 that the `db` attribute is, indeed, required: if the attribute is absent the XML processor can report syntactic model constraint violation even if the element is otherwise lexically well-formed. The document model can also provide additional information not evident without a document model (such as the information on line 4 that the `id` attribute for `purchase` is of XML type `ID`).

1.1.1.2 No built-in meanings or concepts

The area of semantics associated with XML instances is very gray. A document model is but one component used to help describe the semantics of the information found in an instance. While well-formed instances do not have a formal document model, often the names of the constructs used within the instances give hints to the associated semantics. Without a formalism yet available in our community to express semantics in a rigorous fashion, we users of XML do (or should!) capture the semantics of a given vocabulary in prose, whether or not the document model is formalized.

The XML 1.0 Recommendation only describes the behavior required of an XML processor acting on an XML stream, and how it must identify constituent data and provide that data to an application using the processor:

Since there are no formalized semantic description facilities in XML, any XML that is used is not tied to any one particular concept or application. There are no rendition or transformation rules or constructs defined in XML. The only purpose of XML is to unambiguously identify and deliver constituent components of data. There are no inherent meanings or semantics of any kind associated with element types defined in a document model. There are no defined controls for implying any rendering semantics.

Even the `xml:space` attribute allowing for the differentiation of whitespace found in a document is not an aspect of rendering but of information description. The author or modeler of an instance is indicating with this reserved attribute (termed "special" in XML 1.0) the nature of the information and how the whitespace found in the information is to be either preserved or handled by a processor in a default fashion.

Some new users of XML who have a background in a markup language such as HTML often assume a magical association of semantics with element types of the same names they have been exposed to in their prior work. In a web page, they can safely assume that the construct `<p>` will be interpreted as a paragraph or `<em>` as emphasized text. However, this interpretation is solely the purview of the designers of HTML and user agents attempting to conform to the World Wide Web Consortium (W3C)-published semantics. Nothing is imposed by any process when creating a new XML vocabulary that happens to use the same names. Applications using XML processors to access XML information must be instructed how to interpret and implement the desired semantics.

1.1.2 XML Path Language (XPath)

Assuming that we have structured our information using XML, how are we going to talk about (address) what is inside our documents? Locating

information in an XML document is critical to both transforming it and to associating or relating it to other information. When we write stylesheets and use linking languages, we can address components of our information for a processor by our use of the XML Path Language, also called XPath:

- http://www.w3.org/TR/xpath

1.1.2.1 Addressing structured information

The W3C working group responsible for stylesheets collaborated with the W3C working group responsible for the next generation of hyperlinking to produce XPath as a common base for addressing requirements shared by their respective Recommendations. Both groups extend the core XPath facilities to meet the needs they have in each of their domains: the stylesheet group uses XPath as the core of expressions in XSLT; the linking group uses XPath as the core of expressions in the XPointer Recommendation.

In order to address components you have to know the addressing scheme with which the components are arranged. The basis of addressing XML documents is an abstract data model of interlinked nodes arranged hierarchically echoing the tree-shape of the nested elements in an instance. Nodes of different types make up this hierarchy, each node representing the parsed result of a syntactic structure found in the bytes of the XML instance.

This abstraction insulates addressing from the multiple syntactic forms of given XML constructs, allowing us to focus on the information itself and not the syntax used to represent the information.

Note 2:	We see XML documents as a stream or string of bytes that follow the rules of the XML 1.0 Recommendation. Stylesheets do not regard instances in this fashion, and we have to change the way we think of our XML documents in order to successfully work with our information. This leap of understanding ranks high on the list of key aspects of stylesheet writing I needed to internalize before successfully using this technology.

We are given tools to work in the framework provided by the abstraction: a set of data types used to represent values found in the generalization, and a set of functions we use to manipulate and examine those values. The data types include strings, numbers, boolean values and sets of nodes of our information. The functions allow us to cast these values into other data type representations and to return massaged information according to our needs.

1.1.2.2 Addressing identifies a hierarchical position or positions

XPath defines common semantics and syntax for addressing XML-expressed information, and bases these primarily on the hierarchical position of components in the tree. This ordering is referred to as document order in XPath, while in other contexts this is often termed either parse order or depth-first order. Alternatively, we can access an arbitrary location in the tree based on points in the tree having unique identifiers.

We convey XPath addresses in a simple and compact non-XML syntax. This allows us to use an XPath expression as the value of an attribute in an XML vocabulary as in the following examples:

```
01  select="answer"
```

Example 1-3: A simple XPath expression in a `select` attribute

The above attribute value expresses all children named "`answer`" of the current focus element.

```
01  match="question|answer"
```

Example 1-4: An XPath expression in a `match` attribute

The above attribute value expresses a test of an element being in the union of the element types named "`question`" and "`answer`".

The XPath syntax looks a lot like addressing subdirectories in a file system or as part of a Universal Resource Identifier (URI). Multiple steps in a location path are separated by either one or two oblique "`/`" characters. Filters can be specified to further refine the nature of the components of our information being addressed.

```
01  select="question[3]/answer[1]"
```

Example 1-5: A multiple step XPath expression in a `select` attribute

The above example selects only the first "`answer`" child of the third "`question`" child of the focus element.

```
01  select="id('start')//question[@answer='y']"
```

Example 1-6: A more complex XPath expression in a `select` attribute

The above example uses an XPath address identifying some descendants of the element in the instance that has the unique identifier with the value "`start`". Those identified are the question elements whose answer attribute is equal to the string equal to the lower-case letter '`y`'. The value returned is the set of nodes representing the elements meeting the conditions expressed by the address. The address is used in a `select` attribute, thus the XSLT processor is selecting all of the addressed elements for some kind of processing.

1.1.2.3 XPath is *not* a query language

It is important to remember that addressing information is only one aspect of querying information. Other aspects include query operators that massage intermediate results into a final result. While a few operators and functions are available in XSLT to use values identified in documents, these are oriented to string processing, not to complex operations required by some applications.

Note 3:	When query Recommendations are developed, I would hope that the addressing portion is based on XPath as a core, just as with XSLT.

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

What is XSLT? (I)
by G. Ken Holman | Pages: 1, **2**, 3

Print This Article

The Context of XSL Transformations and the XML Path Language (cont'd)

1.1.3 Styling structured information

1.1.3.1 Styling is *transforming* and *formatting* information

Styling is the rendering of information into a form suitable for consumption by a target audience. Because the audience can change for a given set of information, we often need to apply different styling for that information in order to obtain dissimilar renderings in order to meet the needs of each audience. Perhaps some information needs to be rearranged to make more sense for the reader. Perhaps some information needs to be highlighted differently to bring focus to key content.

It is important when we think about styling information to remember that two distinct processes are involved, not just one. First, we must transform the information from the organization used when it was created into the organization needed for consumption. Second, when rendering we must express, whatever the target medium, the aspects of the appearance of the reorganized information.

Consider the flow of information as a streaming process where information is created upstream and processed or consumed downstream. Upstream, in the early stages, we should be expressing the information abstractly, thus preventing any early binding of concrete or final-form concepts. Midstream, or even downstream, we

Table of Contents

1. The context of XSL Transformations and the XML Path Language
•1.1 The XML family of Recommendations
·1.1.1 Extensible Markup Language (XML)
·1.1.2 XML Path Language (XPath)
·1.1.3 Styling structured information
·1.1.4 Extensible Stylesheet Language (XSL)
·1.1.5 Extensible Stylesheet Language Transformations (XSLT)
·1.1.6 Namespaces
·1.1.7 Stylesheet association
•1.2 Transformation data

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

can exploit the information as long as it remains flexible and abstract. Late binding of the information to a final form can be based on the target use of the final product; by delaying this binding until late in the process, we preserve the original information for exploitation for other purposes along the way.

flows
·1.2.1 Transformation from XML to XML
·1.2.2 Transformation from XML to XSL formatting semantics
·1.2.3 Transformation from XML to non-XML
·1.2.4 Three-tiered architectures

It is a common but misdirected practice to model information based on how you plan to use it downstream. It does not matter if your target is a presentation-oriented structure, for example, or a structure that is appropriate for another markup-based system. Modeling practice should focus on both the business reasons and inherent relationships existing in the semantics behind the information being described (as such the vocabularies are then content-oriented). For example, emphasized text is often confused with a particular format in which it is rendered. Where we could model information using a `<b>` element type for eventual rendering in a bold face, we would be better off modeling the information using an `<emph>` element type. In this way we capture the reason for marking up information (that it is emphasized from surrounding information), and we do not lock the downstream targets into only using a bold face for rendering.

Many times the midstream or downstream processes need only rearrange, re-label or synthesize the information for a target purpose and never apply any semantics of style for rendering purposes. Transformation tasks stand alone in such cases, meeting the processing needs without introducing rendering issues.

One caveat regarding modeling content-oriented information is that there are applications where the content-orientation is, indeed, presentation-oriented. Consider book publishing where the abstract content is based on presentational semantics. This is meaningful because there is no abstraction beyond the appearance or presentation of the content.

Consider the customer information in Example 1-1. A web user agent doesn't know how to render an element named `<customer>`. The HTML vocabulary used to render the customer information could be as follows:

```
01  <p>From: <i>(Customer Reference) <b>cust123</b></i>
02  </p>
```

Example 1-7: HTML rendering semantics markup for example

The rendering result would then be as follows, with the rendering user agent interpreting the markup for italics and boldface presentation semantics:

Sponsored By:

Sponsored By:

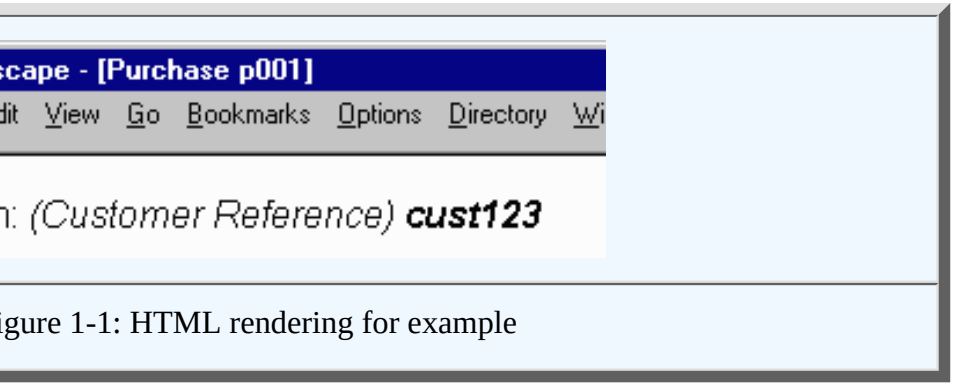


Figure 1-1: HTML rendering for example

The above illustrates these two distinct styling steps: *transforming* the instance of the XML vocabulary into a new instance according to a vocabulary of rendering semantics; and *formatting* the instance of the rendering vocabulary in the user agent.

1.1.3.2 Two W3C Recommendations

In order to meet these two distinct processes in a detached (yet related) fashion, the W3C Working Group responsible for the Extensible Stylesheet Language (XSL) split the original drafts of their work into two separate Recommendations: one for transforming information and the other for rendering information.

The XSL Transformations (XSLT) 1.0 Recommendation describes a vocabulary recognized by an XSLT processor to transform information from an organization in the source file into a different organization suitable for continued downstream processing.

The Extensible Stylesheet Language (XSL) Working Draft describes a vocabulary recognized by a rendering agent to reify abstract expressions of format into a particular medium of presentation.

Both XSLT and XSL are endorsed by members of WSSSL, an association of researchers and developers passionate about the application of markup technologies in today's information technology infrastructure.

1.1.4 Extensible Stylesheet Language (XSL)

When we need to present our structured information in a given medium or different media, we all have common needs for how the result appears and way the result flows through that appearance. The XSL Working Draft describes the current work developing a vocabulary of formatting and flow semantics that can be expressed using an XML model of elements and attributes:

- http://www.w3.org/TR/WD-xsl

1.1.4.1 Formatting and flow semantics vocabulary

This hierarchical vocabulary captures formatting semantics for rendering textual and graphic information in different media. A rendering agent is responsible for interpreting an instance of the vocabulary for a given medium to reify a final result.

This is no different in concept and architecture than using HTML and Cascading Stylesheets (CSS) as a hierarchical vocabulary for rendering a set of information in a web browser. In essence, we are transforming our XML documents into their final display form by transforming instances of our XML vocabularies into instances of a particular rendering vocabulary.

This Working Draft normatively references XSLT as an integral component of XSL. A stylesheet could be written with both the transformation vocabulary and the formatting semantics vocabulary together; it would style an XML instance by rendering the results of transformation. This result need not be serialized in XML syntax; rather, an XSLT/XSL processor can utilize the result of transformation to create a rendered result by interpreting the abstract hierarchy of information without seeing syntax.

1.1.4.2 Target of transformation

When using a formatting semantics vocabulary as the rendering language, the objective for a stylesheet writer is to convert an XML instance of some arbitrary XML vocabulary into an instance of the formatting semantics vocabulary. The result of transformation cannot contain any user-defined vocabulary construct (for example, an address, customer identifier, or purchase order number construct) because the rendering agent would not know what to do with constructs labeled with these foreign, unknown identifiers.

Consider two examples: HTML for rendering in a web browser and XSL for rendering on screen, on paper or audibly. In both cases, the rendering agents only understand the vocabulary expressing their respective formatting semantics and wouldn't know what to do with alien element types defined by the user.

Just as with HTML, a stylesheet writer utilizing XSL for rendering must transform each and every user construct into a rendering construct to direct the rendering agent to produce the desired result. By learning and understanding the semantics behind the constructs of XSL formatting, the stylesheet writer can create an instance of the formatting vocabulary expressing the desired layout of the final result (e.g. area geometry, spacing, font metrics, etc.), with each piece of information in the result coming from either the source data or the stylesheet itself.

Consider once more the customer information in Example 1-1. An XSL rendering agent doesn't know how to render a marked up construct named `<customer>`. The XSL vocabulary used to render the customer information could be as follows:

```
01  <fo:block space-before.optimum="20pt" font-size="20pt">From:
02  <fo:inline-sequence font-style="italic">(Customer Reference)
03  <fo:inline-sequence font-weight="bold">cust123</fo:inline-sequence>
04  </fo:inline-sequence>
05  </fo:block>
```

Example 1-8: XSL rendering semantics markup for example

The rendering result when using the Portable Document Format (PDF) would then be as follows, with an intermediate PDF generation step interpreting the XSL markup for italics and boldface presentation semantics:

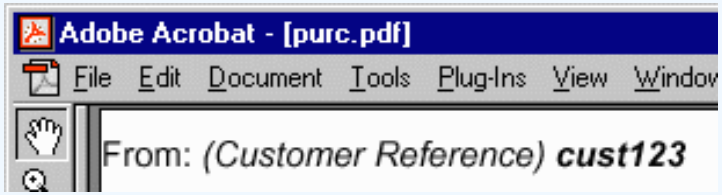


Figure 1-2: XSL rendering for example

The above again illustrates the two distinctive styling steps: *transforming* the instance of the XML vocabulary into a new instance according to a vocabulary of rendering semantics; and *formatting* the instance of the rendering vocabulary in the user agent.

The rendering semantics of much of the XSL vocabulary are device independent, so we can use one set of constructs regardless of the rendering medium. It is the rendering agent's responsibility to interpret these constructs accordingly. In this way, the XSL semantics can be interpreted for print, display, aural or other presentations. There are, indeed, some specialized semantics we can use to influence rendering on particular media, though these are just icing on the cake.

1.1.5 Extensible Stylesheet Language Transformations (XSLT)

We all have needs to transform our structured information when it is not appropriately ordered for a purpose other than how it is created. The XSLT 1.0 Recommendation describes a transformation instruction vocabulary of constructs that can be expressed in an XML model of elements and attributes:

- http://www.w3.org/TR/xslt

1.1.5.1 Transformation by example

We can characterize XSLT from other techniques for transmuting our information by regarding it simply as "Transformation by Example", differentiating many other techniques as "Transformation by Program Logic". This perspective focuses on the distinction that our obligation is not to tell an XSLT processor how to effect the changes we need, rather, we tell an XSLT processor what we want as an end result, and it is the processor's responsibility to do the dirty work.

The XSLT Recommendation gives us a vocabulary for specifying templates that function as "examples of the result". Based on how we instruct the XSLT processor to access the source of the data being transformed, the processor will incrementally build the result by adding the filled-in templates.

We write our stylesheets, or "transformation specifications", primarily with declarative constructs though we can employ procedural techniques if and when needed. We assert the desired behavior of the XSLT processor based on conditions found in our source. We supply examples of how each component of our result is formulated and indicate the conditions of the source that trigger which component is next added to our result. Alternatively we can selectively add components to the result on demand.

Consider once again the customer information in our example purchase order at Example 1-1. An example of the HTML vocabulary supplied to the XSLT processor to produce the markup in Example 1-7 would be:

```
01  <xsl:template match="customer">
02    <p><xsl:text>From: </xsl:text>
03      <i><xsl:text>(Customer Reference) </xsl:text>
04        <b><xsl:value-of select="@db"/></b></i></p>
05  </xsl:template>
```

Example 1-9: Example XSLT template rule for the HTML vocabulary

An example of XSL vocabulary supplied to the XSLT processor to produce the markup in Example 1-8 would be:

```
01  <xsl:template match="customer">
02    <fo:block space-before.optimum="20pt" font-size="20pt">
03      <xsl:text>From: </xsl:text>
04      <fo:inline-sequence font-style="italic">
05        <xsl:text>(Customer Reference) </xsl:text>
06        <fo:inline-sequence font-weight="bold">
07          <xsl:value-of select="@db"/>
08        </fo:inline-sequence></fo:inline-sequence></fo:block>
09  </xsl:template>
```

Example 1-10: Example XSLT template rule for the XSL vocabulary

Where XSLT is similar to other transmutation approaches is that we deal with our information as trees of abstract nodes. We don't deal with the raw syntax of our source data. Unlike these other approaches, however, the primary memory management and information manipulation (node traversal and node creation) is handled by the XSLT processor not by the stylesheet writer. This is a significant difference between XSLT and a transformation programming language or interface like the Document Object Model (DOM), where the programmer is responsible for handling the low-level manipulation of information constructs.

XSLT includes constructs which we use to identify and iterate over structures found in the source information. The information being transformed can be traversed in any order needed and as many times as required to produce the desired result. We can visit source information numerous times if the result of transformation requires that information to be present numerous times.

We users of XSLT don't have the burden of implementing numerous practical algorithms required to present information. The designers of XSLT have specified that such algorithms be implemented within the processor itself, and have enabled us to engage these algorithms declaratively. High-level functions such as sorting and counting are available to us on demand when we need them. Low-level functions such as memory-management, node manipulation and garbage collection are all integral to the XSLT processor.

This declarative nature of the stylesheet markup makes XSLT so very much more accessible to non-programmers than the imperative nature of procedurally-oriented transformation languages. Writing a stylesheet is as simple as using markup to declare the behavior of the XSLT processor, much like HTML is used to declare the behavior of the web browser to paint information on the screen.

The designers have also accommodated the programmer as well as the non-programmer in that there are procedural constructs specified. XSLT is (in theory) "Turing complete", thus any arbitrarily complex algorithm could (theoretically) be implemented using the constructs available. While there will always be a trade-off between extending the processor to implement something internally and writing an elaborate stylesheet to implement something portably, there is sufficient expressive power to implement some algorithmic business rules and semantic processing in the XSLT syntax.

In short, straightforward and common requirements can be satisfied in a straightforward fashion, while unconventional requirements can be satisfied to an extent as well with some programming-styled effort.

Note 4:	Theory aside, the necessarily verbose XSLT syntax dictated by its declarative nature and use of XML syntax makes the coding of some complex algorithms a bit awkward. I have implemented some very complex traversals and content generation with successful results, but with code that could be difficult to maintain (my own valiant, if not always satisfactory, documentation practices notwithstanding).

The designers of XSLT recognized the need to maintain large transformation specifications, and the desire to tap prior accomplishments when writing stylesheets so they have included a number of constructs supporting the management, maintenance and exploitation of existing stylesheets. Organizations can build libraries of stylesheet components for sharing among their colleagues. Stylesheet writers can tweak the results of a transformation by writing shell specifications that include or import other stylesheets known to solve problems they are addressing. Stylesheet fragments can be written for particular vocabulary fragments; these fragments can subsequently be used in concert, as part of an organization's strategy for common information description in numerous markup models.

1.1.5.2 *Not* intended for general purpose XML transformations

It is important to remember that XSLT was designed *primarily for transforming XML vocabularies to the XSL formatting vocabulary*. This doesn't preclude us from using XSLT for other transformation requirements, but it does influence the design of the language and it does constrain some of the functionality from being truly general purpose.

For this reason, the designers *do not* claim XSLT is a general purpose transformation language. However, it is still powerful enough for *most* downstream processing transformation needs, and XSLT stylesheets are often called XSLT transformation scripts because they can be used in many areas not at all related to stylesheet rendering. Consider an electronic commerce environment where transformation is not used for presentation purposes. In this case, the XSLT processor may transform a source instance, which is based on a particular vocabulary, and deliver the results to a legacy application that expects a different vocabulary as input. In other words, we can use XSLT in a non-rendering situation when it doesn't matter what syntax is utilized to represent the content; when only the parsed result of the syntax is material.

An example of using such a legacy vocabulary for the XSLT processor would be:

```
01  <xsl:template match ="customer">
02    <buyer><xsl:value-of select="@db"/></buyer>
03  </xsl:template>
```

Example 1-11: Example XSLT template rule for a legacy vocabulary

The transformation would then produce the following result acceptable to the legacy application:

```
01  <buyer>cust123</buyer>
```

Example 1-12: Example legacy vocabulary for customer information

The designers of XSLT have focused on the results of delivering parsed XML information to a rendering agent, or to some other application employing an XML processor as the means to access information in an

XML instance. The information being delivered represents the parsed result of working with the entire XML instance and, if supplied, the XML document model. The actual markup within the source XML instance is not considered material to the application. All that counts is the result of having processed the XML instance to find the underlying content the actual markup represents.

By focusing on this parsed result for downstream applications, there is little or no regard in an XSLT stylesheet for the actual XML syntax constructs found within the source input documents, or for the actual XML syntax constructs utilized in the resulting output document. This prevents a stylesheet from being aware of such constructs or controlling how such constructs are used. Any transformation requirement that includes "original markup syntax preservation" would not be suited for XSLT transformations.

Note 5:	Is not being able to support "original markup syntax preservation" really a problem? That depends how you regard the original markup syntax used in an XML instance. XML allows you to use various markup techniques to meet identical information representation requirements. If you treat this as merely syntactic sugar for human involvement in the markup process, then it will not be important how information is specifically marked up once it is out of the hands of the human involved. If, however, you are working with transformations where such issues are more than just a sugar coating, and it is necessary to utilize particular constructs based on particular requirements of how the result "looks" in syntactic form, then XSLT will not provide the kind of control you will need.

1.1.5.3 Document model and vocabulary independent

While checking source documents for validity can be very useful for diagnostic purposes, all of the hierarchical relationships of content are based on what is found inside of the instance, not what is found in the document model. The behavior of the stylesheet is specified against the presence of markup in an instance as the *implicit* model, not against the allowed markup prescribed by any *explicit* model. Because of this, an XSLT stylesheet is independent of any Document Type Definition (DTD) or other explicit schema that may have been used to constrain the instance at other stages. This is very handy when working with well-formed XML that doesn't have an explicit document model.

If an explicit document model is supplied, certain information such as attribute types and defaulted values enhance the processor's knowledge of the information found in the input documents. Without this information, the processor can still perform stylesheet processing as long as the absence of the information does not influence the desired results.

Without a reliance on the document model for the instance, we can design a single stylesheet that can process instances of different models. When the models are very similar, much of the stylesheet operates the same way each time and the rest of the stylesheet only processes that which it finds in the sources.

It may be obvious but should be stated for completeness that a given source file can be processed with multiple stylesheets for different purposes. This means, though, that it is possible to successfully process a

source file with a stylesheet designed for an entirely different vocabulary. The results will probably be totally inappropriate, but there is nothing inherent to an instance that ties it to a single stylesheet or a set of stylesheets. Stylesheet designers might well consider how their stylesheets could validate input; perhaps issuing error messages when unexpected content arrives. However, this is a matter of practice and not a constraint.

1.1.5.4 XML source and stylesheet

The input files to an XSLT processor are one or more stylesheet files and one or more source files. The initial inputs are a single stylesheet file and a single source file. Other stylesheet files are assimilated before the first source file is processed. The XML processor will then access other source files according to the first file's XML content. The XSLT processor may then access other source files at any time under stylesheet control.

All of the inputs must be well-formed (but not necessarily valid) XML documents. This precludes using an HTML file following non-XML lexical conventions, but does not rule out processing an Extensible Hypertext Markup Language (XHTML) file as an input. Many users of existing HTML files that are not XML compliant will need to manipulate or transform them; all that is needed to use XSLT for this is a preprocess to convert existing Standard Generalized Markup Language (SGML) markup conventions into XML markup conventions.

XHTML can be created from HTML using a handy free tool on the W3C site: http://www.w3.org/People/Raggett/tidy/. This tool corrects whatever improperly coded HTML it can and flags any that it cannot correct. When the output is configured to follow XML lexical conventions, the resulting file can be used as an input to the XSLT processor.

1.1.5.5 Validation unnecessary (but convenient)

That an XSLT processor need not incorporate a validating XML processor to do its job does not minimize the importance of source validation when developing a stylesheet. Often when working incrementally to develop a stylesheet by simultaneously working on the test source file and stylesheet algorithm, time can be lost by inadvertently introducing well-formed but invalid source content. Because there is no validation in the XSLT processor, all well-formed source will be processed without errors, producing a result based on the data found. The first reaction of the stylesheet writer is often that a problem has been introduced in the stylesheet logic, when in fact the stylesheet works fine for the intended source data. The real problem is that the source data being used isn't as intended.

Note 6:	Personally, I run a separate post-process source file validation after running the source file through a given stylesheet. While I am examining the results of stylesheet processing, the post process determines whether or not the well-formed file validates against the model to which I'm designing the stylesheet. When anomalies are seen I can check the validation for the possible source of a problem before diagnosing the stylesheet itself.

1.1.5.6 Multiple source files possible

The first source file fed to the XSLT processor defines the first abstract tree of nodes the stylesheet uses.

The stylesheet may access arbitrary other source files, or even itself as a source file, to supplement the information found in the primary file. The names of these supplementary resources can be hardwired into the stylesheet, passed to the stylesheet as a parameter, or the stylesheet can find them in the source files.

A separate node tree represents every resource accessed as a source file, each with its own scope of unique node identifiers and global values. When a given resource is identified more than once as a source file, the XSLT processor creates only a single representation for that resource. In this way a stylesheet is guaranteed to work unambiguously with source information.

1.1.5.7 Stylesheet supplements source

A given transformation result does not necessarily obtain all of its information from the source files. It is often (almost always) necessary to supplement the source with boilerplate or other hardwired information. The stylesheet can add any arbitrary information to the result tree as it builds the result tree from information found in the source trees.

A stylesheet can be the synthesis of the primary file and any number of supplemental files that are included or imported by the main file. This provides powerful mechanisms for sharing and exploiting fragments of stylesheets in different scenarios.

1.1.5.8 Extensible language design supplements processing

The "X" in XSLT stands for "Extensible" for a reason: the designers have built-in conforming techniques for accessing non-conforming facilities requested by a stylesheet writer that may or may not be available in the XSLT processor interpreting the stylesheet. A conforming processor may or may not support such extensions and is only obliged to accommodate error and fallback processing in such a way that a stylesheet writer can reconcile the behavior if needed.

An XSLT processor can implement extension instructions, functions, serialization conventions and sorting schemes that provide functionality beyond what is defined in XSLT 1.0, all accessed through standardized facilities.

A stylesheet writer must not rely on any extension facilities if the XSLT processor being used for the stylesheet is not known or is outside of the stylesheet writer's control. If an end-user base utilizes different brands of XSLT processors, and the stylesheet needs to be portable across all processors, only the standardized facilities can be used.

Standardized presence-testing and fallback facilities can be used by the stylesheet writer to accommodate

the ability of a processor to act on extension facilities used in the stylesheet.

1.1.5.9 Abstract structure result

In the same way our stylesheets are insulated from the syntax of our source files, our stylesheets are insulated from the syntax of our result.

We do not focus on the syntax of the file to be produced by the XSLT processor; rather, we create a result tree of abstract nodes, which is similar to the tree of abstract nodes of our input information. Our examples of transformation (converted to nodes from our stylesheet) are added to the result hierarchy as nodes, not as syntax. Our objective as XSLT transformation writers is to create a result node tree that may or may not be serialized externally as markup syntax.

The XSLT processor is not obliged to externalize the result tree if the processor is integral to some process interpreting the result tree for other purposes. For example, an XSL rendering agent may embed an XSLT processor for interpreting the inputs to produce the intermediate hierarchy of XSL rendering vocabulary to be reified in a given medium. In such cases, serializing the intermediate tree in syntax is not material to the process of rendering (though having the option to serialize the hierarchy is a useful diagnostic tool).

The stylesheet writer has little or no control over the constructs chosen by the XSLT processor for serializing the result tree. There are some behaviors the stylesheet can request of the processor, though the processor is not obliged to respect the requests. The stylesheet can request a particular output method be used for the serialization and, if supported, the processor guarantees the final result complies with the lexical requirements of that method.

Note 7:	It is possible to coerce the XSLT processor to violate the lexical rules through certain stylesheet controls that I personally avoid using at all costs. For every XML and HTML instance construct (not including the document model syntax constructs) there are proper XSLT methodologies to follow, though not always as compact as coercing the processor.

The abstract nature of the node trees representing the input source and stylesheet instances and the hands-off nature of serializing the abstract result node tree are the primary reasons that source tree original markup syntax preservation cannot be supported.

The design of the language does, however, support the serialization of the result tree in such a way as not to require the XSLT processor to maintain the result tree in the abstract form. For example, the processor can instantly serialize the start of an element as soon as the element content of the result is defined. There is no need to maintain, nor is there any ability in the stylesheet to add to, the start of an element once the stylesheet begins supplying element content.

The XSLT 1.0 Recommendation defines three output methods for lexically reifying the abstract result tree as serialized syntax: XML conventions, HTML conventions, and simple text conventions. An XSLT

processor can be extended to support custom serialization methods for specialized needs.

1.1.5.10 Result-tree-oriented objective

This result abstraction impacts how we design our stylesheets. We have to always remember that the result of transformation is created in result parse order, thus allowing the XSLT processor to immediately serialize the result without maintaining the result for later purposes.

The examples of transformation that we include in our stylesheet already represent examples of the nodes that we want added to the result tree, but we must ensure these examples are triggered to be added to the result tree in result parse order, otherwise we will not get the desired result.

We can peruse and traverse our source files in any predictable order we need to produce the result, but we can only produce the result tree once and then only in result tree parse order. It is often difficult to change traditional perspectives of transformation that focus on the source tree, yet we must look at XSLT transformations focused on the result tree.

The predictable orders we traverse the source trees are not restricted to only source tree parse order (also called document order). Information in the source trees can be ignored or selectively processed. The order of the result tree dictates the order in which we must access our source trees.

Note 8:	I personally found this required orientation difficult to internalize, having been focused on the creation of my source information long before addressing issues of transforming the sources to different results. Understanding this orientation is key to quickly producing results using XSLT.

It is not, however, an XSLT processor implementation constraint to serially produce the result tree. This is an important distinction in the language design that supports parallelism. An XSLT processor supporting parallelism can simultaneously produce portions of the result tree provided only that the end result is created as if it were produced serially.

1.1.6 Namespaces

To successfully use and distinguish element types in our instances as being from given vocabularies, the Namespaces in XML Recommendation gives us means to preface our element type names to make them unique. The Recommendation and the following widely-read discussion document describe the precepts for using this technique:

- http://www.w3.org/TR/REC-xml-names
- http://www.megginson.com/docs/namespaces/namespace-questions.html

1.1.6.1 Vocabulary distinction

It would be unreasonable to mandate that all document models have mutually unique element type names. We design our document models with our own business requirements and our own naming conventions; so do other users. A W3C working group developing vocabularies has its own conventions and requirements; so do other committees. An XML-based application knowing that an instance is using element types from only a single vocabulary can easily distinguish all elements by the name, since each element type is declared in the model by its name.

But what happens when we need to create an XML instance that contains element types from more than one vocabulary? If all the element types are uniquely named then we could guess the vocabulary for a given element by its name. But if the same name is used in more than one vocabulary, we need a technique to avoid ambiguity. Using cryptically compressed or unmanageably elongated element type names to guarantee uniqueness would make XML difficult to use and would only delay the problem to the point that these weakened naming conventions would still eventually result in vocabulary collisions.

Note 9:	Enter the dreaded namespaces: a Recommendation undeserving of its sullied reputation. This is a powerful, yet very simple technique for disambiguating element type names in vocabularies. Perhaps the reputation spread from those unfamiliar with the requirements being satisfied. Perhaps concerns were spread by those who made assumptions about the values used in namespace declarations. As unjustified as it is, evoking namespaces unnecessarily (and unfortunately) strikes fear in many people. It is my goal to help the reader understand that not only are namespaces easy to define and easy to use, but that they are easy to understand and are not nearly as complex as others have believed.

The Namespaces in XML Recommendation describes a technique for exploiting the established uniqueness of Uniform Resource Identifier (URI) values under the purview of the Internet Engineering Task Force (IETF). We users of the Internet accept the authority of the registrar of Internet domain names to allot unique values to organizations, and it is in our best interest to not arrogate or usurp values allotted to others as our own. We can, therefore, assume a published URI value belongs to the owner of the domain used as the basis of the value. The value is not a Uniform Resource Locator (URL), which is a URI that identifies an actual addressed location on the Internet; rather, the URI is being used merely as a unique string value.

To set the stage for how these URI values are used, consider an example of two vocabularies that could easily be used together in an XML instance: the Scalable Vector Graphics (SVG) vocabulary and the Mathematical Markup Language (MathML). In SVG the `<set>` element type is used to scope a value for reference by descendent elements. In MathML the `<set>` element type defines a set in the mathematical sense of a collection.

Remembering that names in XML follow rigid lexical constraints, we pick out of thin air a prefix we use to distinguish each element type from their respective vocabulary. The prefix we choose is *not* mandated by any organization or any authority; in our instances we get to choose any prefix we wish. We should, however, make the prefix meaningful or we will obfuscate our information, so let's choose in this example

to distinguish the two element types as `<svg:set>` and `<math:set>`. Note that making the prefix short is a common convention supporting human legibility, and using the colon ":" separating the prefix from the rest of the name is prescribed by the Namespaces in XML recommendation.

While we are talking about names, let's not forget that some Recommendations utilize the XML name lexical construct for other purposes, such as naming facilities that may be available to a processor. We get to use this namespace prefix we've chosen on these names to guarantee uniqueness, just as we have done on the names used to indicate element types.

1.1.6.2 URI value association

But having the prefix is not enough because we haven't yet guaranteed global identity or singularity by a short string of name characters; to do so we must associate the prefix with a globally unique URI before we use that prefix. Note that we are unable to use a URI directly as a prefix because the lexical constraints on a URI are looser than those of an XML name; the invalid XML name characters in a URI would cause an XML processor to balk.

We assert the association between a namespace prefix and a namespace URI by using a namespace declaration attribute as in the following examples:

- `xmlns:svg="http://www.w3.org/2000/svg-20000629"`
- `xmlns:math="http://www.w3.org/1998/Math/MathML"`

As noted earlier, the prefix we choose is arbitrary and can be any lexically valid XML name. The prefix is discarded by the namespace-aware processor, and is immaterial to the application using the names; it is only a syntactic shortcut to get at the associated URI. The associated URI supplants the prefix in the internal representation of the name value and the application can distinguish the names by the new composite name that would have been illegal in XML syntax. There is no convention for documenting a namespace qualified name using its associated URI, but one way to perceive the uniqueness is to consider our example as it might be internally represented by an application:

- `<{http://www.w3.org/2000/svg-20000629}set>`
- `<{http://www.w3.org/1998/Math/MathML}set>`

The specification of a URI instead of a URL means that the namespace-aware processor will never look at the URI as a URL to accomplish its work. There never need be any resource available at the URI used in a namespace declaration. The URI is just a string and its value is used only as a string and the fact that there may or may not be any resource at the URL identified by the URI is immaterial to namespace processing. The URI does not identify the location of a schema, or a DTD or any file whatsoever when used by a namespace aware processor.

Note 10:	Perhaps some of the confusion regarding namespaces is rooted in the overloading of the namespace URI by some Recommendations. These Recommendations require that the URI represent a URL where a particular resource is located, fetched, and utilized to some purpose. This behavior is outside the scope of namespaces and is mandated solely by the Recommendations that require it. Practice has, however, indicated an end-user-friendly convention regarding the URI used in namespace declarations. The W3C has placed a documentation file at every URL represented by a namespace URI. Requesting the resource at the URL returns an HTML document discussing the namespace being referenced, perhaps a few pointer documents to specifications or user help information, and any other piece of helpful information deemed suitable for the public consumption. This convention should help clear up many misperceptions about the URI being used to obtain some kind of machine-readable resource or schema, though it will not dispel the misperception that there needs to be some resource of some kind at the URL represented by a namespace URI.

So now a processor can unambiguously distinguish an element's type as being from a particular vocabulary by knowing the URI associated with the vocabulary. Our choice of prefix is arbitrary and of no relevance. The URI we have associated with the prefix used in a namespace-qualified XML name (often called a QName) informs the processor of the identity of the name. Our choice of prefix is used and then discarded by the processor, while the URI persists and is the basis of namespace-aware processing. We have achieved uniqueness and identity in our element type names and other XML names in a succinct legible fashion without violating the lexical naming rules of XML.

1.1.6.3 Namespaces in XSL and XSLT

Namespaces identify different constructs for the processors interpreting XSL formatting specifications and XSLT stylesheets.

An XSL rendering agent responsible for interpreting an XSL formatting specification will recognize those constructs identified with the `http://www.w3.org/1999/XSL/Format` namespace. Note that the year value used in this URI value is not used as a version indictor; rather, the W3C convention for assigning namespace URI values incorporates the year the value was assigned to the working group.

An XSLT processor responsible for interpreting an XSLT stylesheet recognizes instructions and named system properties using the `http://www.w3.org/1999/XSL/Transform` namespace. An XSLT processor will not recognize using an archaic value for working draft specifications of XSLT.

XSLT use namespace-qualified names to identify extensions that implement non-standardized facilities. A number of kinds of extensions can be defined in XSLT including functions, instructions, serialization methods, sort methods and system properties.

The XT XSLT processor written by James Clark is an example of a processor implementing extension facilities. XT uses the `http://www.jclark.com/xt` namespace to identify the extension constructs it implements. Remembering that this is a URI and not a URL, you will not find any kind of resource or file when using this value as a URL.

We also use our own namespaces in an XSLT stylesheet for two other purposes. We need to specify the namespaces of the elements and attributes of our result if the process interpreting the result relies on the vocabulary to be identified. Furthermore, our own non-default namespaces distinguish internal XSLT objects we include in our stylesheets. Each of these will be detailed later where such constructs are described.

1.1.7 Stylesheet association

When we wish to associate with our information one or more preferred or suitable stylesheet resources geared to process that information, the W3C stylesheet association Recommendation describes the syntax and semantics for a construct we can add to our XML documents:

- `http://www.w3.org/TR/xml-stylesheet`

1.1.7.1 Relating documents to their stylesheets

XML information in its authored form is often not organized in an appropriate ordering for consumption. A stylesheet association processing instruction is used at the start of an XML document to indicate to the recipient which stylesheet resources are to be used when reading the contents of that document.

The recipient is not obliged to use the resources referenced and can choose to examine the XML using any stylesheet or transformation process they desire by ignoring the preferences stated within. Some XML applications ignore the stylesheet association instruction entirely, while others choose to steadfastly respect the instruction without giving any control to the recipient. A flexible application will let the recipient choose how they wish to view the content of the document.

The designers of this specification adopted the same semantics of the `<LINK>` construct defined in the HTML 4.0 recommendation:

- `<LINK REL="stylesheet">`
- `<LINK REL="alternate stylesheet">`

1.1.7.2 Ancillary markup

A processing instruction is ancillary to the XML document model constraining the creation and validation of an instance. Therefore, we do not have to model the presence of this construct when we design our document model. Any instance can have any number of stylesheet associations added into the document during or after creation, or even removed, without impacting on the XML content itself.

An application respecting this construct will process the document content with the stylesheet before delivering the content to the application logic. Two cases of this are the use of a stylesheet for rendering to a browser canvas and the use of a transformation script at the front end of an e-commerce application.

The following two examples illustrate stylesheet associations that, respectively, reference an XSL resource and a Cascading Stylesheet (CSS) resource:

```
01  <?xml-stylesheet href="fancy.xsl" type="text/xsl"?>
```

Example 1-13: Associating an XSL stylesheet

```
01  <?xml-stylesheet href="normal.css" type="text/css"?>
```

Example 1-14: Associating a CSS stylesheet

The following example naming the association for later reference and indicating that it is not the primary stylesheet resource is less typical, but is allowed for in the specification:

```
01  <?xml-stylesheet alternate="yes" title="small"
02                   href="small.xsl" type="text/xsl"?>
```

Example 1-15: Alternative stylesheet association

A URL that does not include a reference to another resource, but rather is defined exclusively by a local named reference, specifies a stylesheet resource that is located inside the XML document being processed, as in the following example:

```
01  <?xml-stylesheet href="#style1" type="text/xsl"?>
```

Example 1-16: Associating an internal stylesheet

The Recommendation designers expect additional schemes for linking stylesheets and other processing scripts to XML documents to be defined in future specifications.

Note 11:	Embedding stylesheet association information in an XML document and using the XML processing instruction to do so are both considered stopgap measures by the W3C. This Recommendation cautions readers that no precedents are set by employing these makeshift techniques and that urgency dictated their choice. Indeed, there is some question as to the appropriateness of tying processing to data so tightly, and we will see what considered approaches become available to us in the future.

Pages: 1, **2**, 3

Next Page

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

The State of Streaming Media
Find it on the O'Reilly Network

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND

What is XSLT? (I)
by G. Ken Holman | Pages: 1, 2, **3**

Print This Article

The Context of XSL Transformations and the XML Path Language (cont'd)

1.2 Transformation data flows

Here we look at the interactions between some of the Recommendations we focus on by examining how our information flows through processes engaging or supporting the technologies.

1.2.1 Transformation from XML to XML

As we will see when looking at the data model, the normative behavior of XSLT is to transform an XML source into an abstract hierarchical result. We can request that result to be serialized into an XML file, thus we achieve XML results from XML sources:

Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

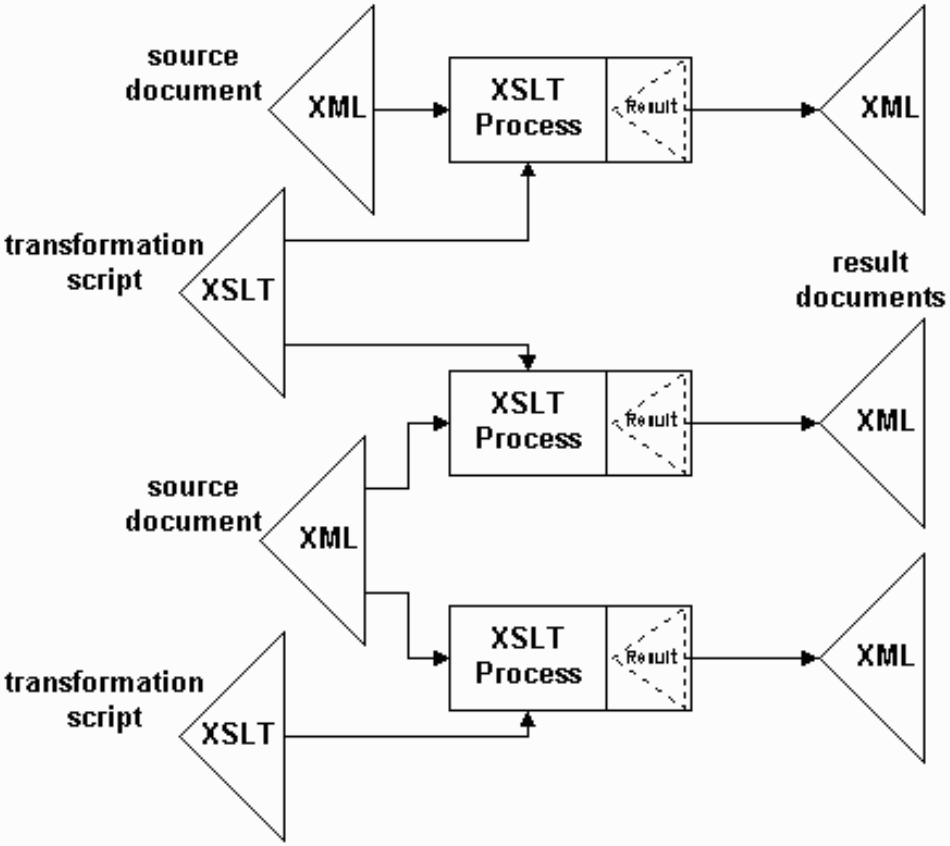


Figure 1-3: Transformation from XML to XML

An XSLT stylesheet can be applied to more than one XML document, each stylesheet producing a possibly (usually) different result. Nothing in XSLT inherently ties the stylesheet to a single instance, though the stylesheet writer can employ techniques to abort processing based on processing undesirable input.

An XML document can have more than one XSLT stylesheet applied, each stylesheet producing a possible (usually) different result. Even when stylesheet association indicates an author's preference for a stylesheet to use for processing, tools should provide the facility to override the preference with the reader's preference for a stylesheet. Nothing in XML prevents more than a single stylesheet to be applied.

Sponsored By:

Sponsored By:

Note 12:	In all cases in this chapter the depictions show the normative result of the XSLT processor's as the dotted triangle attached to the process rectangle. This serves to remind the reader that the serialization of the result into an XML file is a separate task, one that is the responsibility of the XSLT processor and not the stylesheet writer. In all diagrams, the left-pointing triangle represents a hierarchically-marked up document such as an XML or HTML document. This convention stems from considering the apex of the hierarchy at the left, with the sub-elements nesting within each other towards the lowest leaves of the hierarchy at the right of the triangle. Processes are depicted in rectangles, while arbitrary data files of some binary or text form are depicted in parallelograms. Other symbols representing screen display, print and auditory output are drawn with (hopefully) obvious shapes.

1.2.2 Transformation from XML to XSL formatting semantics

When the result tree is specified to utilize the XSL formatting vocabulary, the normative behavior of an XSL processor incorporating an XSLT processor is to interpret the result tree. This interpretation reifies the semantics expressed in the constructs of the result tree to some medium, be it pixels on a screen, dots on paper, sound through a synthesis device, or another medium that makes sense for presentation.

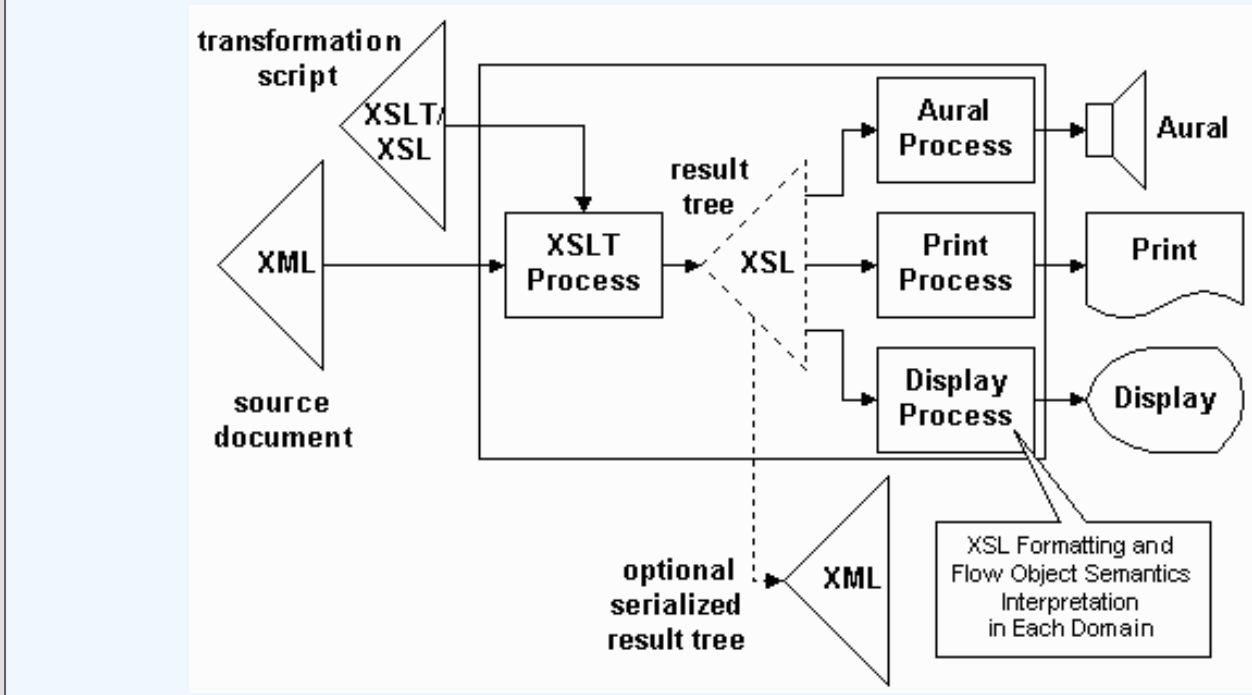


Figure 1-4: Transformation from XML to XSL Formatting Semantics

Without employing extension techniques or supplemental documentation, the stylesheets used in this scenario contain *only* the transformation vocabulary and the resulting formatting vocabulary. There are no other element types from other vocabularies in the result, including from the source vocabulary. For example, rendering processors would not inherently know what to do with an element of type custnbr representing a customer number; it is the stylesheet writer's responsibility to transform the information into information recognized by the rendering agent.

There is no obligation for the rendering processor to serialize the result tree created during transformation. The feature of serializing the result tree to XML syntax is, however, quite useful as a diagnostic tool, revealing to us what we really asked to be rendered instead of what we thought we were asking to be rendered when we saw incorrect results. There may also be performance considerations of taking the reified result tree in XML syntax and rendering it in other media without incurring the overhead of performing the transformation repeatedly.

1.2.3 Transformation from XML to non-XML

An XSLT processor may choose to recognize the stylesheet writer's desire to serialize a non-XML representation of the result tree:

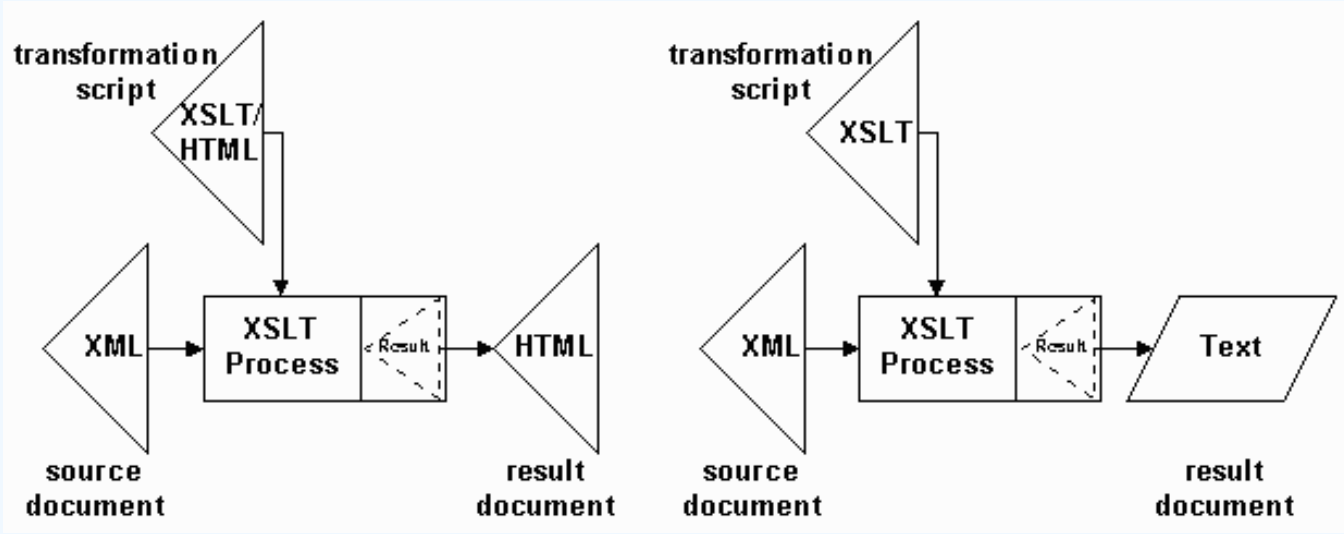


Figure 1-5: Transformation from XML to Aware Non-XML

The XSLT Recommendation documents two non-XML tree serialization methods that can be requested by the stylesheet writer. When the processor offers serialization, it is *only* obliged to reify the result using XML lexical and syntax rules, and *may* support producing output following either HTML lexical and syntax rules or simple text.

1.2.3.1 HTML lexical and syntactic conventions

Internet web browsers are specific examples of the generic HTML user agent. User agents are typically designed for instances of HTML following the precursor to XML: the Standard Generalized Markup Language (SGML) lexical conventions. Certain aspects of the HTML document model also dictate syntactic shortcuts available when working with SGML.

While some more recently developed user agents will accept XML lexical conventions, thus accepting Extensible Hypertext Markup Language (XHTML) output from an XSLT processor, older user agents will not. Some of these user agents will not accept XML lexical conventions for empty elements, while some require SGML syntax minimization techniques to compress certain attribute specifications.

Additionally, user agents recognize a number of general entity references as built-in characters supporting accented letters, the non-breaking space, and other characters from public entity sets defined or used by the designers of HTML. An XSLT processor recognizes the use of these characters in the result tree and serializes them using the assumed built-in general entities.

1.2.3.2 Text lexical conventions

An XSLT processor can be asked to serialize only the #PCDATA content of the entire result tree, resulting in a file of simple text without employing any markup techniques. All text is represented by the characters' individual values, even those characters sensitive to XML interpretation.

Note 13:	I use the text method often for synthesizing MSDOS batch files. By walking through my XML source I generate commands to act on resources identified therein, thus producing an executable batch file tailored to the information.

1.2.3.3 Arbitrary binary and custom lexical conventions

Many of our legacy systems or existing applications expect information to follow custom lexical conventions according to arbitrary rules. Often, this format is raw binary not following textual lexical patterns. We are usually obliged to write custom programs and transformation applications to convert our XML information to these non-standardized formats due to their binary or non-structured natures.

XSLT can play a role even here where the target format is neither structured, nor text, nor in any format anticipated by the designers of the Recommendation. We do have a responsibility to fill in a critical piece of the formula described below, but we can leverage this single effort in the equation to allow us and our colleagues to continue to use W3C Recommendations with our XML data.

Not using XSLT to produce custom output

Consider first the scenario without using XSLT where we must write individual XML-aware applications to accommodate our information vocabularies. For each of our vocabularies we need *separate* programs to convert to the common custom format required by the application. This incurs programming resources to accommodate any and every change to our vocabularies in order to meet the new inputs to satisfy the same custom output.

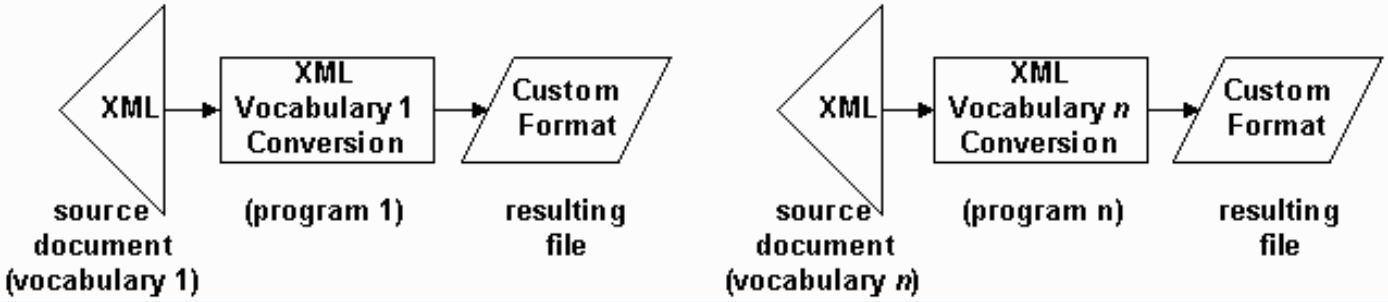


Figure 1-6: Accommodating multiple inputs with different XML vocabularies

Using XSLT to produce custom output

If, however, we focus on the custom output instead of focusing on our vocabulary inputs, we can leverage a single investment in programming across all of our vocabularies. Moreover, by being independent of the vocabulary used in the source, we can accommodate any of our or others' vocabularies we may have to deal with in the future.

The approach involves us creating our own custom markup language based on a critical analysis of the target custom format to distill the semantics of how information is represented in the resulting file. These semantics can be expressed using an XML vocabulary whose elements and attributes engage the features and functions of the resulting format. We must not be thinking of our source XML vocabularies, rather, our focus is entirely on the semantics of what exactly makes up our target custom format. Let's refer to this custom format's XML vocabulary we divine from our analysis as the Custom Vocabulary Markup Language (CVML).

Using our programming resources we can then write a single transformation application responsible for interpreting XML instances of CVML to produce a file following the custom format. This transformation application could be written using the Document Object Model (DOM) as a basis for tree-oriented access to the information. Alternatively, a SAX-based application can interpret the instances to produce the outputs if the nature of CVML lends itself to that orientation. The key is that regardless of how instances of CVML are created, the interpretation of CVML markup to produce an output file never changes. Our *one* CVML Instance Interpreter application can produce *any* custom format output file expressible in the CVML semantics.

Getting back to our own or others' XML vocabularies, we have now reduced the problem to XML instance transformation. Our objective is simplified to produce XML instances of CVML from instances of our many input XML vocabularies. This is a classical XSLT situation and we need only write XSLT stylesheets combining the XSLT instructions with CVML as the result vocabulary. Our investment in XSLT for our colleagues is leveraged by the CVML Instance Interpreter so that they can now take their XML and use stylesheets to produce the binary or custom lexical format.

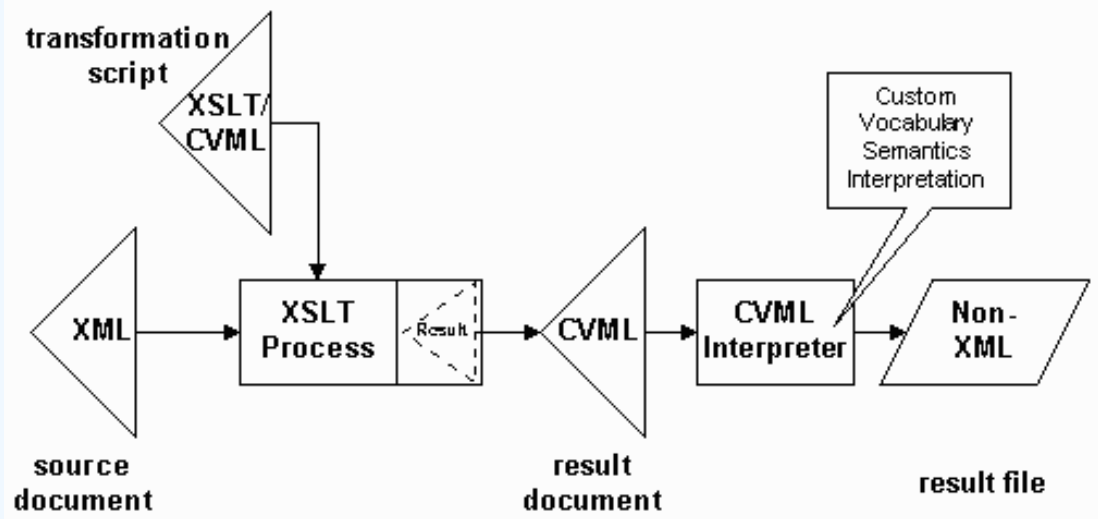


Figure 1-7: Transformation from XML to an arbitrary format

This approach separates the awareness of the lexical and syntactic requirements of the custom output format from the numerous stylesheets we write for all of our possible input XML vocabularies. Our colleagues use XSLT just as they would with HTML or XSL as a result vocabulary. They leverage the single investment in producing the custom format by using the CVML Interpreter to serialize the results of their transformations to produce the files designed for other applications. This, in turn, leverages the investment in learning and using XSLT in the organization.

Taking this two steps further

First, the "X" in XSLT represents the word "extensible" and result tree serialization is one of the areas where we can extend an XSLT processor's functionality. This allows us to implement non-standard vendor-specific or application-specific output serialization methods and engage these facilities in a standard manner. As with all extension mechanisms in XSLT, the trigger is the use of an XML namespace recognized by the XSLT processor implementing the extension:

```
01  xmlns:prefix="processor-recognized-URI"
02  <xsl:output method="prefix:serialization-method-name"/>
```

Example 1-17: Using namespaces to specify an extension serialization method

Comment:
- the namespace declaration attribute on line 1 must be somewhere in the element or the ancestry of the instruction on line 2

Using the same semantics described for the outboard CVML Interpreter program depicted in Figure 1-7, this translation facility can be incorporated into the XSLT processor itself as an inboard extension. The code itself may be

directly portable based on the nature of how the outboard program is written. Such an extended processor would directly emit the custom format without reifying the intermediate structure (though this would be convenient for diagnostic purposes):

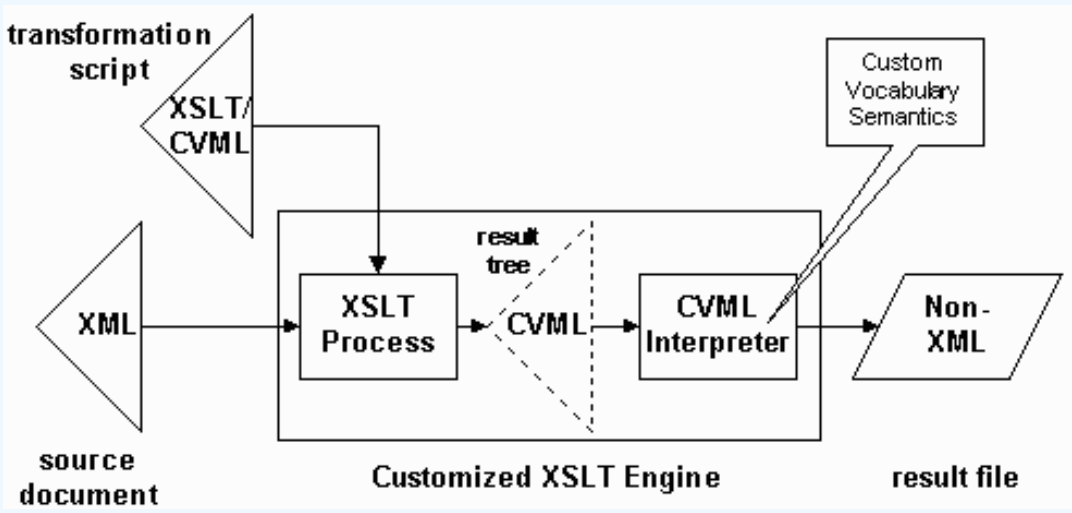


Figure 1-8: Built-in Transformation from XML to Arbitrary Non-XML

The XT XSLT processor implements an extension serialization method named NXML for "non-XML":

```
01  xmlns:prefix="http://www.jclark.com/xt"
02  <xsl:output method="prefix:nxml"/>
```

Example 1-18: Using the XT namespace to specify the NXML extension serialization method

Comment:
- the namespace declaration attribute on line 1 must be somewhere in the element or the ancestry of the instruction on line 2

Second, this extensibility opens up the opportunity to use an XSLT processor as a front-end to any application that can be modified to access the result tree. The intermediate result tree of CVML is not serialized externally; rather, it is fed directly to the application and the application interprets the internal representation of the content that would have been serialized to a custom format. Time is saved by not serializing the result tree and having the application parse the reified file back into a memory representation; performance is enhanced by the application directly accessing the result of transformation.

When generalized, a vendor's non-XML-based application can use this approach to accommodate arbitrary customers' XML vocabularies merely by writing W3C conforming XSLT stylesheets as the "interpretation specification". Some XSLT processors can build a DOM representation of result tree or deliver the result tree as Simple API for XML (SAX) events, thus giving an application developer standardized interfaces to the transformed information expressed

using the application's custom semantics vocabulary. The developer's programming is then complete and the vendor accommodates each customer vocabulary with an appropriate stylesheet for translation to the application semantics.

1.2.4 Three-tiered architectures

A three-tiered architecture can meet technical and business objectives by delivering structured information to web browsers by using XSLT on the host, or on the user agent, or even on both.

Considering technical issues first, the server can distribute the processing load to XML/XSLT-aware user agents by delivering a combination of the stylesheet and the source information to be transformed on the recipient's platform. Alternatively, the server can perform the transformations centrally to accommodate those user agents supporting only HTML or HTML/CSS vocabularies:

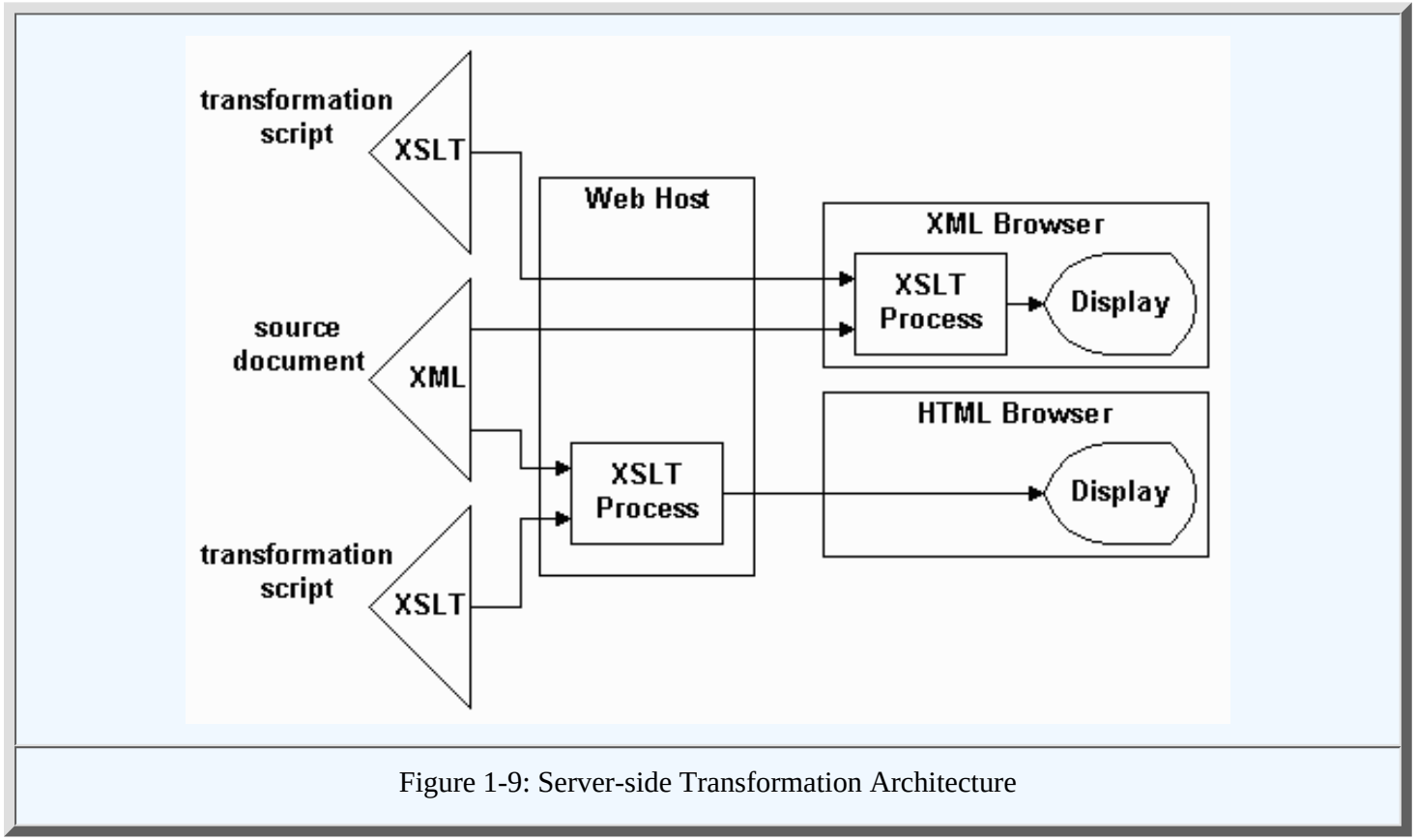


Figure 1-9: Server-side Transformation Architecture

There may be good business reasons to selectively deliver richly-marked-up XML to the user agent or to arbitrarily transform XML to HTML on the server regardless of the user agent capabilities. Even if it is technically possible to send semantically-rich information in XML, protecting your intellectual property by hiding the richness behind the security of a "semantic firewall" must be considered. Perhaps there are revenue opportunities by only delivering a richly marked-up rendition of your information to your customers. Perhaps you could even scale the richness to

differing levels of utility for customers who value your information with different granularity or specificity, while preserving the most detailed normative version of the data away of view.

Lastly, there are no restrictions to using two XSLT processes: one on the server to translate our organization's rich markup into an arbitrary delivery-oriented markup. This delivery markup, in turn, is translated using XSLT on the user agent for consumption by the operator. This approach can reduce bandwidth utilization and increase distributed processing without sacrificing privacy.

Note 14:	There is no consensus in our XML community that semantic firewalls are a "good thing". Peers of mine preach that the World Wide Web must always be a semantic web with rich markup processed in a distributed fashion among user agents being de rigueur. Personally, I do not subscribe to this point of view. We have the flexibility to weigh the technical and business perspectives of our customers' needs for our information, our own infrastructure and processing capabilities, and our own commercial and privacy concerns. We can choose to "dumb down" our information for consumption, and the installed base of user agents supporting presentation-oriented semantic-less HTML can be the perfect delivery vehicle to protect these concerns of ours.

This is a prose version of an excerpt from the book "Practical Transformation Using XSLT and XPath" (Eighth Edition ISBN 1-894049-05-5 at the time of this writing) published by Crane Softwrights Ltd., written by G. Ken Holman; this excerpt was edited by Stan Swaren, and reviewed by Dave Pawson.

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Pre-conference workshops: 25th July
Conference: 26th&27th July
Holiday Inn London, Victoria
Register Now

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

Print This Article

Getting started with XSLT and XPath

by G. Ken Holman
August 23, 2000

Getting started with XSLT and XPath

Examining working stylesheets can help us understand how we use XSLT and XPath to perform transformations. This article first dissects some example stylesheets before introducing basic terminology and design principles.

Table of Contents

2. Getting started with XSLT and XPath
- 2.1 Stylesheet examples
  - 2.1.1 Some simple examples
  - 2.1.2 Some more complex examples
- 2.2 Syntax basics -- stylesheets, templates, instructions
  - 2.2.1 Explicitly declared stylesheets
  - 2.2.2 Implicitly declared stylesheets

2.1 Stylesheet examples

Let's first look at some example stylesheets using two implementations of XSLT 1.0 and XPath 1.0: the XT processor from James Clark, and the third web release of Internet Explorer 5's MSXML Technology Preview.

These two processors were chosen merely as examples of, respectively, standalone and browser-based XSLT/XPath implementations, without prejudice to other conforming implementations. The code samples only use syntax conforming to XSLT 1.0 and XPath 1.0 recommendations

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

and will work with any conformant XSLT processor.

·2.2.3 Stylesheet requirements
ons and literal
ts
es and template
ches to
stylesheet design

Note: The current (4/14/2000) Internet Explorer 5 *production* release supports only an archaic experimental dialect of XSLT based on an early working draft of the recommendation. The examples in this book *will not* run on the production release of IE5. The production implementation of the old dialect is described in http://msdn.microsoft.com/xml/XSLGuide/conformance.asp.

2.1.1 Some simple examples

Consider the following XML file `hello.xml` obtained from the XML 1.0 Recommendation and modified to declare an associated stylesheet:

```
01  <?xml version="1.0"?>
02  <?xml-stylesheet type="text/xsl" href="hello.xsl"?>
03  <greeting>Hello world.</greeting>
```

Example 2-1: The first sample instance in XML 1.0 (modified)

We will use this simple file as the source of information for our transformation. Note that the stylesheet association processing instruction in line 2 refers to a stylesheet with the name "`hello.xsl`" of type XSL. Recall that an XSLT processor is *not* obliged to respect the stylesheet association preference, so let us first use a standalone XSLT processor with the following stylesheet `hellohtm.xsl`:

```
01  <?xml version="1.0"?><!--hellohtm.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03  <html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04        xsl:version="1.0">
05   <head><title>Greeting</title></head>
06   <body><p>Words of greeting:<br/>
07     <b><i><u><xsl:value-of select="greeting"/></u></i></b>
08     </p></body>
09  </html>
```

Example 2-2: An implicitly-declared simple stylesheet

This file looks like a simple XHTML file: an XML file using the HTML vocabulary. Indeed, it is just that, but we are allowed to inject into the instance XSLT instructions using the prefix for the XSLT vocabulary declared on line 3. We can use any XML file as an XSLT stylesheet provided it declares the XSLT vocabulary within and indicates the version of XSLT being used. Any prefix can be used for XSLT instructions, though

convention often sees `xsl:` as the prefix value.

Line 7 contains the only XSLT instruction in the instance. The `xsl:value-of` instruction uses an XPath expression in the `select=` attribute to calculate a string value from our source information. XPath views the source hierarchy using parent/child relationships. The XSLT processor's initial focus is the root of the document, which is considered the parent of the document element. Our XPath expression value "`greeting`" selects the child named "`greeting`" from the current focus, thus returning the value of the document element named "`greeting`" from the instance.

Using an MS-DOS command-line invocation to execute the standalone processor, we see the following result:

```
01  X:\samp>xt hello.xml hellohtm.xsl hellohtm.htm
02  X:\samp>type hellohtm.htm
03  <html>
04  <head>
05  <title>Greeting</title>
06  </head>
07  <body>
08  <p>Words of greeting:<br>
09  <b><i><u>Hello world.</u></i></b>
10  </p>
11  </body>
12  </html>
13
14  X:\samp>
```

Example 2-3: Explicit invocation of Example 2-2

Note how the end result contains a mixture of the stylesheet markup and the source instance content, without any use of the XSLT vocabulary. The processor has recognized the use of HTML by the name of the document element and has engaged SGML lexical conventions.

The SGML lexical conventions are evidenced on line 8 where the `<br>` empty element has been serialized without the XML lexical convention for the closing delimiter. This corresponds to line 6 of our stylesheet in Example 2-2 where this element is marked up as `<br/>` according to XML rules. Our inputs are always XML but the XSLT processor may recognize the output as being HTML and serialize the result following SGML rules.

Consider next the following explicitly-declared XSLT file `hello.xsl` to produce XML output using the HTML vocabulary, thus the output is serialized as XHTML:

```
01  <?xml version="1.0"?><!--hello.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03
04  <xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
05                 version="1.0">
06
07  <xsl:output method="xml" omit-xml-declaration="yes"/>
08
09  <xsl:template match="/">
10      <b><i><u><xsl:value-of select="greeting"/></u></i></b>
11  </xsl:template>
12
13  </xsl:transform>
```

Example 2-4: An explicitly-declared simple stylesheet

This file explicitly declares the document element of an XSLT stylesheet with the requisite XSLT namespace and version declarations. Line 7 declares the output to follow XML lexical conventions and that the XML declaration is to be omitted from the serialized result. Lines 9 through 11 declare the content of the result that is added when the source information position matches the XPath expression in the `match=` attribute on line 9. The value of "`/`" matches the root of the document, hence, this refers to the XSLT processor's initial focus.

The result we specify on line 10 wraps our source information in the HTML elements without the boilerplate used in the previous example. Line 13 ends the formal specification of the stylesheet content.

Using an MS-DOS command-line invocation to execute the XT processor we see the following result:

```
01  X:\samp>xt hello.xml hello.xsl hello.htm
02
03  X:\samp>type hello.htm
04  <b><i><u>Hello world.</u></i></b>
05  X:\samp>
```

Example 2-5: Explicit invocation of Example 2-4

Using a non-XML-aware browser to view the resulting HTML in Example 2-5 we see the following on the canvas (the child window is opened using the View/Source menu item):

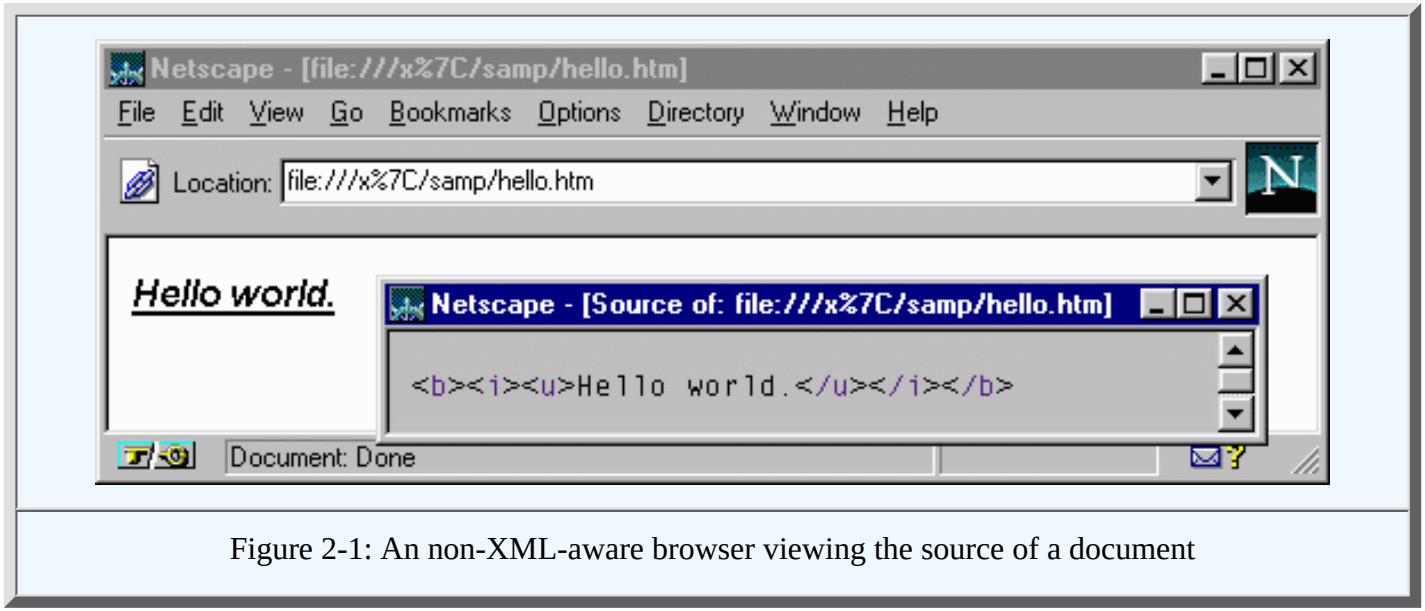


Figure 2-1: An non-XML-aware browser viewing the source of a document

Using an XML-aware browser recognizing the W3C stylesheet association processing instruction in Example 2-1, the canvas is painted with the HTML resulting from application of the stylesheet (the child window is opened using the View/Source menu item):

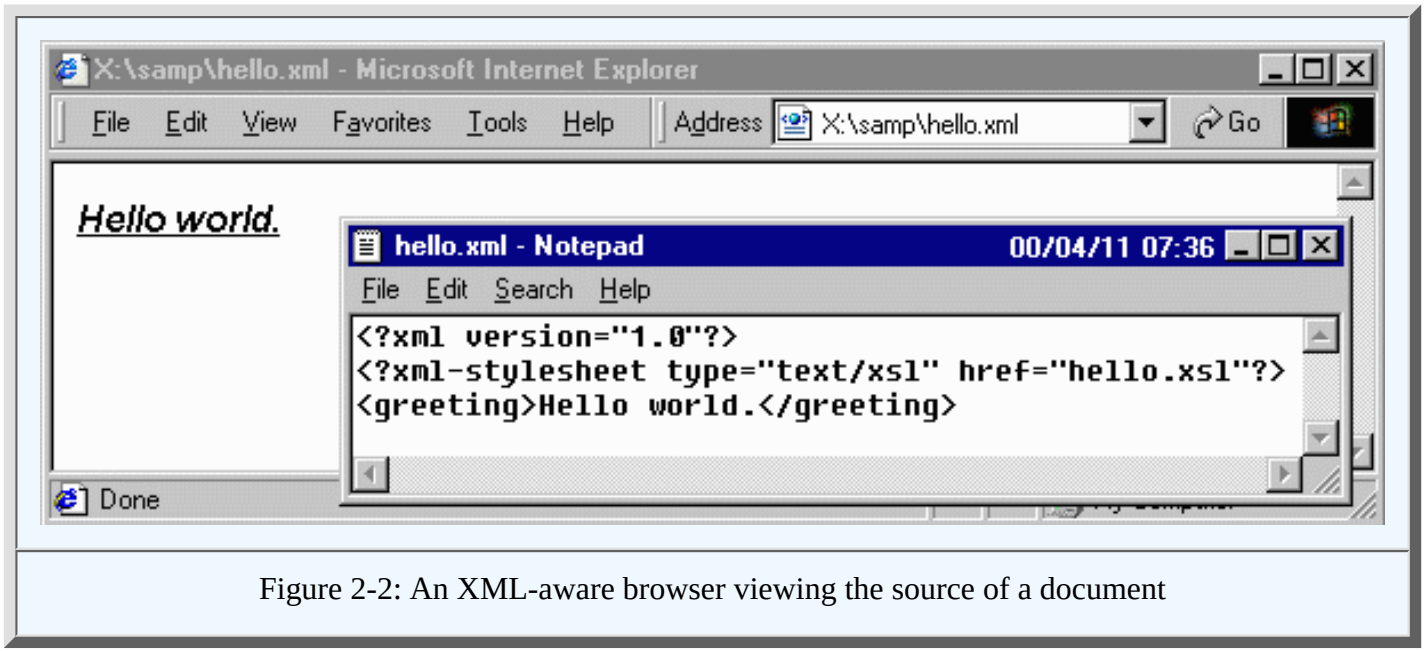


Figure 2-2: An XML-aware browser viewing the source of a document

The canvas content matches what the non-XML browser rendered in Figure 2-1. Note that View/Source

displays the raw XML source and not the transformed XHTML result of applying the stylesheet.

Note:	I found it very awkward when first using browser-based stylesheets to diagnose problems in my stylesheets. Without access to the intermediate results of transformation, it is often impossible to ascertain the nature of the faulty HTML generation. One of the free resources found on the Crane Softwrights Ltd. web site is a script for standalone command-line invocation of the MSXML XSLT processor. This script is useful for diagnosing problems by revealing the result of transformation. This script has also been used extensively by some to create static HTML snapshots of their XML for delivery to non-XML-aware browsers.

Pages: **1**, 2, 3

Click Here Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate... THAWTE www.thawte.com

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Pre-conference workshops: 25th July
Conference: 26th&27th July
Holiday Inn London, Victoria
Register Now

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

SSH Tunneling Secures 802.11
In the Wireless DevCenter

Print This Article

Getting started with XSLT and XPath

by G. Ken Holman
August 23, 2000

Getting started with XSLT and XPath

Examining working stylesheets can help us understand how we use XSLT and XPath to perform transformations. This article first dissects some example stylesheets before introducing basic terminology and design principles.

Table of Contents

2. Getting started with XSLT and XPath
- 2.1 Stylesheet examples
  - 2.1.1 Some simple examples
  - 2.1.2 Some more complex examples
- 2.2 Syntax basics -- stylesheets, templates, instructions
  - 2.2.1 Explicitly declared stylesheets
  - 2.2.2 Implicitly declared stylesheets

2.1 Stylesheet examples

Let's first look at some example stylesheets using two implementations of XSLT 1.0 and XPath 1.0: the XT processor from James Clark, and the third web release of Internet Explorer 5's MSXML Technology Preview.

These two processors were chosen merely as examples of, respectively, standalone and browser-based XSLT/XPath implementations, without prejudice to other conforming implementations. The code samples only use syntax conforming to XSLT 1.0 and XPath 1.0 recommendations

XML
Products
Tools
Training
O'REILLY xml.com

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

and will work with any conformant XSLT processor.

·2.2.3 Stylesheet requirements
ons and literal
ts
es and template
ches to
stylesheet design

Note: The current (4/14/2000) Internet Explorer 5 *production* release supports only an archaic experimental dialect of XSLT based on an early working draft of the recommendation. The examples in this book *will not* run on the production release of IE5. The production implementation of the old dialect is described in http://msdn.microsoft.com/xml/XSLGuide/conformance.asp.

2.1.1 Some simple examples

Consider the following XML file `hello.xml` obtained from the XML 1.0 Recommendation and modified to declare an associated stylesheet:

```
01  <?xml version="1.0"?>
02  <?xml-stylesheet type="text/xsl" href="hello.xsl"?>
03  <greeting>Hello world.</greeting>
```

Example 2-1: The first sample instance in XML 1.0 (modified)

We will use this simple file as the source of information for our transformation. Note that the stylesheet association processing instruction in line 2 refers to a stylesheet with the name "`hello.xsl`" of type XSL. Recall that an XSLT processor is *not* obliged to respect the stylesheet association preference, so let us first use a standalone XSLT processor with the following stylesheet `hellohtm.xsl`:

```
01  <?xml version="1.0"?><!--hellohtm.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03  <html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04        xsl:version="1.0">
05   <head><title>Greeting</title></head>
06   <body><p>Words of greeting:<br/>
07     <b><i><u><xsl:value-of select="greeting"/></u></i></b>
08     </p></body>
09  </html>
```

Example 2-2: An implicitly-declared simple stylesheet

This file looks like a simple XHTML file: an XML file using the HTML vocabulary. Indeed, it is just that, but we are allowed to inject into the instance XSLT instructions using the prefix for the XSLT vocabulary declared on line 3. We can use any XML file as an XSLT stylesheet provided it declares the XSLT vocabulary within and indicates the version of XSLT being used. Any prefix can be used for XSLT instructions, though

convention often sees `xsl:` as the prefix value.

Line 7 contains the only XSLT instruction in the instance. The `xsl:value-of` instruction uses an XPath expression in the `select=` attribute to calculate a string value from our source information. XPath views the source hierarchy using parent/child relationships. The XSLT processor's initial focus is the root of the document, which is considered the parent of the document element. Our XPath expression value "`greeting`" selects the child named "`greeting`" from the current focus, thus returning the value of the document element named "`greeting`" from the instance.

Using an MS-DOS command-line invocation to execute the standalone processor, we see the following result:

```
01  X:\samp>xt hello.xml hellohtm.xsl hellohtm.htm
02  X:\samp>type hellohtm.htm
03  <html>
04  <head>
05  <title>Greeting</title>
06  </head>
07  <body>
08  <p>Words of greeting:<br>
09  <b><i><u>Hello world.</u></i></b>
10  </p>
11  </body>
12  </html>
13
14  X:\samp>
```

Example 2-3: Explicit invocation of Example 2-2

Note how the end result contains a mixture of the stylesheet markup and the source instance content, without any use of the XSLT vocabulary. The processor has recognized the use of HTML by the name of the document element and has engaged SGML lexical conventions.

The SGML lexical conventions are evidenced on line 8 where the `<br>` empty element has been serialized without the XML lexical convention for the closing delimiter. This corresponds to line 6 of our stylesheet in Example 2-2 where this element is marked up as `<br/>` according to XML rules. Our inputs are always XML but the XSLT processor may recognize the output as being HTML and serialize the result following SGML rules.

Consider next the following explicitly-declared XSLT file `hello.xsl` to produce XML output using the HTML vocabulary, thus the output is serialized as XHTML:

```
01  <?xml version="1.0"?><!--hello.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03
04  <xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
05                 version="1.0">
06
07  <xsl:output method="xml" omit-xml-declaration="yes"/>
08
09  <xsl:template match="/">
10      <b><i><u><xsl:value-of select="greeting"/></u></i></b>
11  </xsl:template>
12
13  </xsl:transform>
```

Example 2-4: An explicitly-declared simple stylesheet

This file explicitly declares the document element of an XSLT stylesheet with the requisite XSLT namespace and version declarations. Line 7 declares the output to follow XML lexical conventions and that the XML declaration is to be omitted from the serialized result. Lines 9 through 11 declare the content of the result that is added when the source information position matches the XPath expression in the `match=` attribute on line 9. The value of "`/`" matches the root of the document, hence, this refers to the XSLT processor's initial focus.

The result we specify on line 10 wraps our source information in the HTML elements without the boilerplate used in the previous example. Line 13 ends the formal specification of the stylesheet content.

Using an MS-DOS command-line invocation to execute the XT processor we see the following result:

```
01  X:\samp>xt hello.xml hello.xsl hello.htm
02
03  X:\samp>type hello.htm
04  <b><i><u>Hello world.</u></i></b>
05  X:\samp>
```

Example 2-5: Explicit invocation of Example 2-4

Using a non-XML-aware browser to view the resulting HTML in Example 2-5 we see the following on the canvas (the child window is opened using the View/Source menu item):

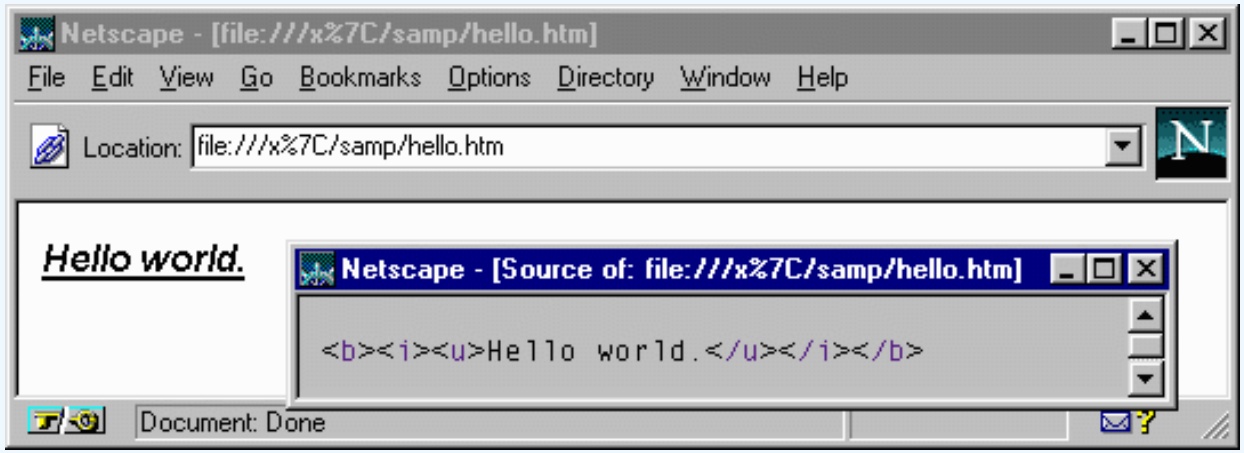


Figure 2-1: An non-XML-aware browser viewing the source of a document

Using an XML-aware browser recognizing the W3C stylesheet association processing instruction in Example 2-1, the canvas is painted with the HTML resulting from application of the stylesheet (the child window is opened using the View/Source menu item):

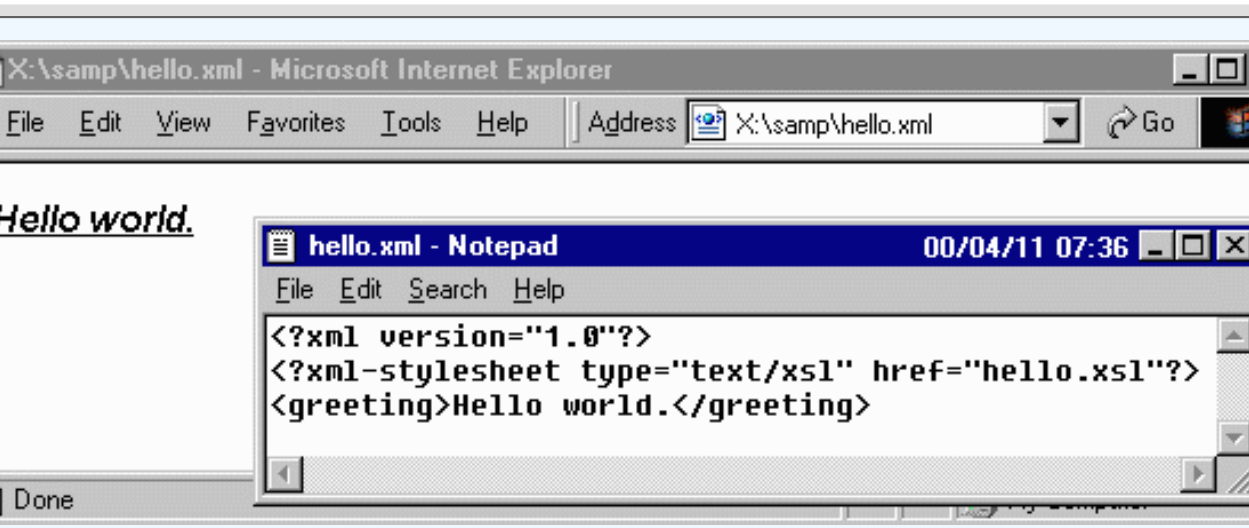


Figure 2-2: An XML-aware browser viewing the source of a document

The canvas content matches what the non-XML browser rendered in Figure 2-1. Note that View/Source

displays the raw XML source and not the transformed XHTML result of applying the stylesheet.

Note:	I found it very awkward when first using browser-based stylesheets to diagnose problems in my stylesheets. Without access to the intermediate results of transformation, it is often impossible to ascertain the nature of the faulty HTML generation. One of the free resources found on the Crane Softwrights Ltd. web site is a script for standalone command-line invocation of the MSXML XSLT processor. This script is useful for diagnosing problems by revealing the result of transformation. This script has also been used extensively by some to create static HTML snapshots of their XML for delivery to non-XML-aware browsers.

Pages: **1**, 2, 3

Next Page

Click Here Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate... THAWTE www.thawte.com

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com xml from the inside out

XML europe 2001 CONFERENCE & EXPOSITION 21-25 May 2001 • Berlin, Germany www.xmleurope.com

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS

Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS

Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

Getting started with XSLT and XPath
by G. Ken Holman | Pages: 1, **2**, 3

Print This Article

2.1.2 Some more complex examples

The following more complex examples are meant merely as illustrations of some of the powerful facilities and techniques available in XSLT. These samples expose concepts such as variables, functions, and process control constructs a stylesheet writer uses to effect the desired result, but does not attempt any tutelage in their use.

Note:	This subsection can be skipped entirely, or, for quick exposure to some of the facilities available in XSLT and XPath, only briefly reviewed. In the associated narratives, I've avoided the precise terminology that hasn't yet been introduced and I overview the stylesheet contents and processor behaviors in only broad terms. Subsequent subsections of this chapter review some of the basic terminology and design approaches. I hope not to frighten the reader with the complexity of these examples, but it is important to realize that there are more complex operations than can be illustrated using our earlier three-line source file example. The complexity of your transformations will dictate the complexity of the stylesheet facilities being engaged. Simple transformations can be performed quite simply using XSLT, but not all of us have to meet only simple requirements.

The following XML source information in `prod.xml` is used to produce two very dissimilar renderings:

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

```
01  <?xml version="1.0"?><!--prod.xml-->
02  <!DOCTYPE sales [
03  <!ELEMENT sales ( products, record )> <!--sales information-->
04  <!ELEMENT products ( product+ )>         <!--product record-->
05  <!ELEMENT product ( #PCDATA )>      <!--product information-->
06  <!ATTLIST product id ID #REQUIRED>
07  <!ELEMENT record ( cust+ )>                <!--sales record-->
08  <!ELEMENT cust ( prodsale+ )>     <!--customer sales record-->
09  <!ATTLIST cust num CDATA #REQUIRED>     <!--customer number-->
10  <!ELEMENT prodsale ( #PCDATA )>     <!--product sale record-->
11  <!ATTLIST prodsale idref IDREF #REQUIRED>
12  ]>
13  <sales>
14    <products><product id="p1">Packing Boxes</product>
15              <product id="p2">Packing Tape</product></products>
16    <record><cust num="C1001">
17              <prodsale idref="p1">100</prodsale>
18              <prodsale idref="p2">200</prodsale></cust>
19            <cust num="C1002">
20              <prodsale idref="p2">50</prodsale></cust>
21            <cust num="C1003">
22              <prodsale idref="p1">75</prodsale>
23              <prodsale idref="p2">15</prodsale></cust></record>
24  </sales>
```

Example 2-6: Sample product sales source information

Lines 2 through 11 describe the document model for the sales information. Lines 14 and 15 summarize product description information and have unique identifiers according to the `ID`/`IDREF` rules. Lines 16 through 23 summarize customer purchases (product sales), each entry referring to the product having been sold by use of the `idref=` attribute. Not all customers have been sold all products.

Consider the following two renderings of the same data using two orientations, each produced with different stylesheets:

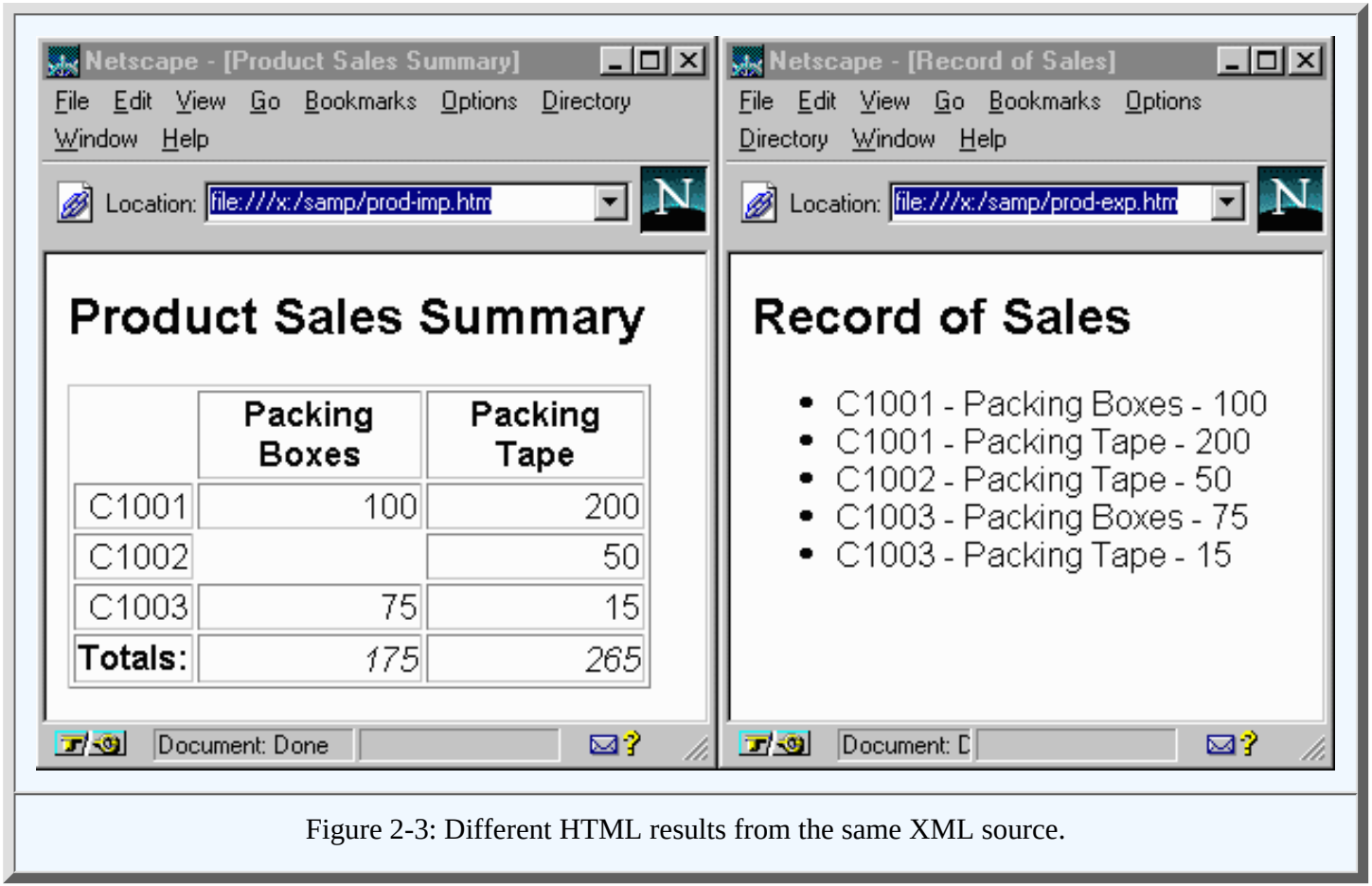


Figure 2-3: Different HTML results from the same XML source.

Note how the same information is projected into a table orientation on the left canvas and a list orientation on the right canvas. The one authored order is delivered in two different presentation orders. Both results include titles from boilerplate text not found in the source. The table information on the left includes calculations of the sums of quantities in the columns, generated by the stylesheet and not present explicitly in the source.

The implicit stylesheet `prod-imp.xsl` is an XHTML file utilizing the XSLT vocabulary for instructions to fill in the one result template by pulling data from the source:

```
01  <?xml version="1.0"?><!--prod-imp.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03  <html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04        xsl:version="1.0">
05    <head><title>Product Sales Summary</title></head>
06    <body><h2>Product Sales Summary</h2>
07      <table summary="Product Sales Summary" border="1">
08                                              <!--list products-->
09        <th align="center">
10            <xsl:for-each select="//product">
11              <td><b><xsl:value-of select="."/></b></td>
12            </xsl:for-each></th>
13                                             <!--list customers-->
14        <xsl:for-each select="/sales/record/cust">
15          <xsl:variable name="customer" select="."/>
16          <tr align="right"><td><xsl:value-of select="@num"/></td>
17            <xsl:for-each select="//product">   <!--each product-->
18              <td><xsl:value-of select="$customer/prodsale
19                                        [@idref=current()/@id]"/>
20              </td></xsl:for-each>
21          </tr></xsl:for-each>
22                                                  <!--summarize-->
23        <tr align="right"><td><b>Totals:</b></td>
24            <xsl:for-each select="//product">
25              <xsl:variable name="pid" select="@id"/>
26              <td><i><xsl:value-of
27                       select="sum(//prodsale[@idref=$pid])"/></i>
28              </td></xsl:for-each></tr>
29      </table>
30    </body></html>
```

Example 2-7: Tabular presentation of the sample product sales source information

Recall that a stylesheet is oriented according to the desired result, producing the result in result parse order. The entire document is an HTML file whose document element begins on line 3 and ends on line 30. The XSLT namespace and version declarations are included in the document element. The naming of the document element as "html" triggers the default use of HTML result tree serialization conventions. Lines 5 and 6 are fixed boilerplate information for the mandatory `<title>` element.

Lines 7 through 29 build the result table from the content. A single header row `<th>` is generated in lines 9 through 12, with the columns of that row generated by traversing all of the `<product>` elements of the source. The focus

moves on line 11 to each `<product>` source element in turn and the markup associated with the traversal builds each `<td>` result element. The content of each column is specified as "`.`", which for an element evaluates to the string value of that element.

Having completed the table header, the table body rows are then built, one at a time traversing each `<cust>` child of a `<record>` child of the `<sales>` child of the root of the document, according to the XPath expression "`/sales/record/cust`". The current focus moves to the `<cust>` element for the processing on lines 15 through 21. A local scope variable is bound on line 15 with the tree location of the current focus (note how this instruction uses the same XPath expression as on line 11 but with a different result). A table row is started on line 16 with the leftmost column calculated from the `num=` attribute of the `<cust>` element being processed.

The stylesheet then builds in lines 17 through 20 a column for each of the same columns created for the table header on line 10. The focus moves to each product in turn for the processing of lines 18 through 20. Each column's value is then calculated with the expression "`$customer/prodsale[@idref=current()/@id]`", which could be expressed as follows "from the customer location bound to the variable `$customer`, from all of the `<prodsale>` children of that customer, find that child whose `idref=` attribute is the value of the `id=` attribute of the focus element." When there is no such child, the column value is empty and processing continues. As many columns are produced for a body row as for the header row and our output becomes perfectly aligned.

Finally, lines 23 through 28 build the bottom row of the table with the totals calculated for each product. After the boilerplate leftmost column, line 24 uses the same "`//product`" expression as on lines 10 and 17 to generate the same number of table columns. The focus changes to each product for lines 25 through 28. A local scope variable is bound with the focus position in the tree. Each column is then calculated using a built-in function as the sum of all `<prodsale>` elements that reference the column being totaled. The XPath designers, having provided the `sum()` function in the language, keep the stylesheet writer from having to implement complex counting and summing code; rather, the writer merely declares the need for the summed value to be added to the result on demand by using the appropriate XPath expression.

The file `prod-exp.xsl` is an explicit XSLT stylesheet with a number of result templates for handling source information:

```
01  <?xml version="1.0"?><!--prod-exp.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04                  version="1.0">
05
06  <xsl:template match="/">                          <!--root rule-->
07    <html><head><title>Record of Sales</title></head>
08      <body><h2>Record of Sales</h2>
09        <xsl:apply-templates select="/sales/record"/>
10      </body></html></xsl:template>
11
```

```
12  <xsl:template match="record">   <!--processing for each record-->
13    <ul><xsl:apply-templates/></ul></xsl:template>
14
15  <xsl:template match="prodsale">   <!--processing for each sale-->
16    <li><xsl:value-of select="../@num"/>   <!--use parent's attr-->
17        <xsl:text> - </xsl:text>
18        <xsl:value-of select="id(@idref)"/>     <!--go indirect-->
19        <xsl:text> - </xsl:text>
20        <xsl:value-of select="."/></li></xsl:template>
21
22  </xsl:stylesheet>
```

Example 2-8: List-oriented presentation of the sample product sales source information

The document element on line 3 includes the requisite declarations of the language namespace and the version being used in the stylesheet. The children of the document element are the template rules describing the source tree event handlers for the transformation. Each event handler associates a template with an event trigger described by an XPath expression.

Lines 6 through 10 describe the template rule for processing the root of the document, as indicated by the "`/`" trigger in the `match=` attribute on line 6. The result document element and boilerplate is added to the result tree on lines 7 and 8. Line 9 instructs the XSLT processor in `<xsl:apply-templates>` to visit all `<record>` element children of the `<sales>` document element, as specified in the `select=` attribute. For each location visited, the processor pushes that location through the stylesheet, thus triggering the template of result markup it can match for each location.

Lines 12 and 13 describe the result markup when matching a `<record>` element. The focus moves to the `<record>` element being visited. The template rule on line 13 adds the markup for the HTML unordered list `<ul>` element to the result tree. The content of the list is created by instructing the processor to visit all children of the focus location (implicitly by not specifying any `select=` attribute) and apply the templates of result markup it triggers for each child. The only children of `<record>` are `<cust>` elements.

The stylesheet does not provide any template rule for the `<cust>` element, so built-in template rules automatically process the children of each location being visited in turn. Implicitly, then, our source information is being traversed in the depth-first order, visiting the locations in parse order and pushing each location through any template rules that are then found in the stylesheet. The children of the `<cust>` elements are `<prodsale>` elements.

The stylesheet does provide a template rule in lines 15 through 20 to handle a `<prodsale>` element when it is pushed, so the XSLT processor adds the markup triggered by that rule to the result. The focus changes when the template rule handles it, thus, lines 16, 18, and 20 each pull information relative to the `<prodsale>` element, respectively: the parent's `num=` attribute (the `<cust>` element's attribute); the string value of the target element being pointed to by the `<prodsale>` element's `idref=` attribute (indirectly obtaining the `<product>` element's value); and the value of the `<prodsale>` element itself.

Click Here Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate... THAWTE www.thawte.com

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Pre-conference workshops: 25th July
Conference: 26th&27th July
Holiday Inn London, Victoria
Register Now

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

Designing JSP Custom Tag Libraries
Find it at ONJava.com

Getting started with XSLT and XPath
by G. Ken Holman | Pages: 1, 2, **3**

Print This Article

Getting started with XSLT and XPath (III)

2.2 Syntax basics: Stylesheets, Templates, Instructions

Next we'll look at some basic terminology both helpful in understanding the principles of writing an XSLT stylesheet and recognizing the constructs used therein. This section is not meant as tutelage for writing stylesheets, but only as background information, nomenclature, and practice guidelines.

Note:	I use two pairs of diametric terms not used as such in the XSLT Recommendation itself: explicit/implicit stylesheets and push/pull design approaches. Students of my instructor-led courses have found these distinctions helpful even though they are not official terms. Though these terms are documented here with apparent official status, such status is not meant to be conferred.

2.2.1 Explicitly declared stylesheets

An explicitly declared XSLT stylesheet is comprised of a distinct wrapper element containing the stylesheet specification. This wrapper element must be an XSLT instruction named either `stylesheet` or `transform`, thus it must be qualified by the prefix associated with the XSLT namespace URI. This wrapper element is the document element in a standalone stylesheet, but may in other cases be embedded inside an XML document.

XML
Products
Tools
Training
O'REILLY xml.com

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

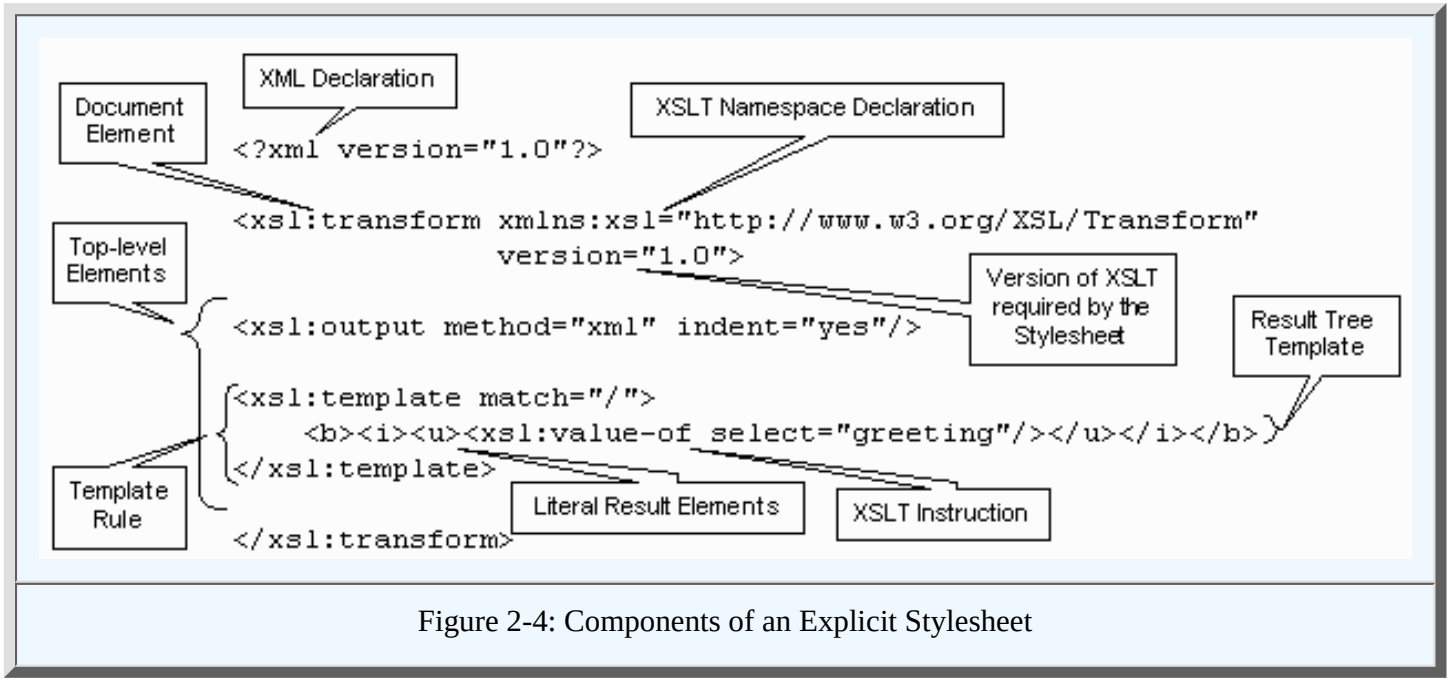


Figure 2-4: Components of an Explicit Stylesheet

The XML declaration is consumed by the XML processor embedded within the XSLT processor, thus the XSLT processor never sees it. The wrapper element must include the XSLT namespace and version declarations for the element to be recognized as an instruction.

The children of the wrapper element are the top-level elements, comprised of global constructs, serialization information, and certain maintenance instructions. Template rules supply the stylesheet behavior for matching source tree conditions. The content of a template rule is a result tree template containing both literal result elements and XSLT instructions.

The example above has only a single template rule, that being for the root of the document.

2.2.2 Implicitly declared stylesheets

The simplest kind of XSLT stylesheet is an XML file implicitly representing the entire outcome of transformation. The result vocabulary is arbitrary, and the stylesheet tree forms the template used by the XSLT processor to build the result tree. If no XSLT or extension instructions are found therein, the stylesheet tree becomes the result tree. If instructions are present, the processor replaces the instructions with the outcomes of their execution.

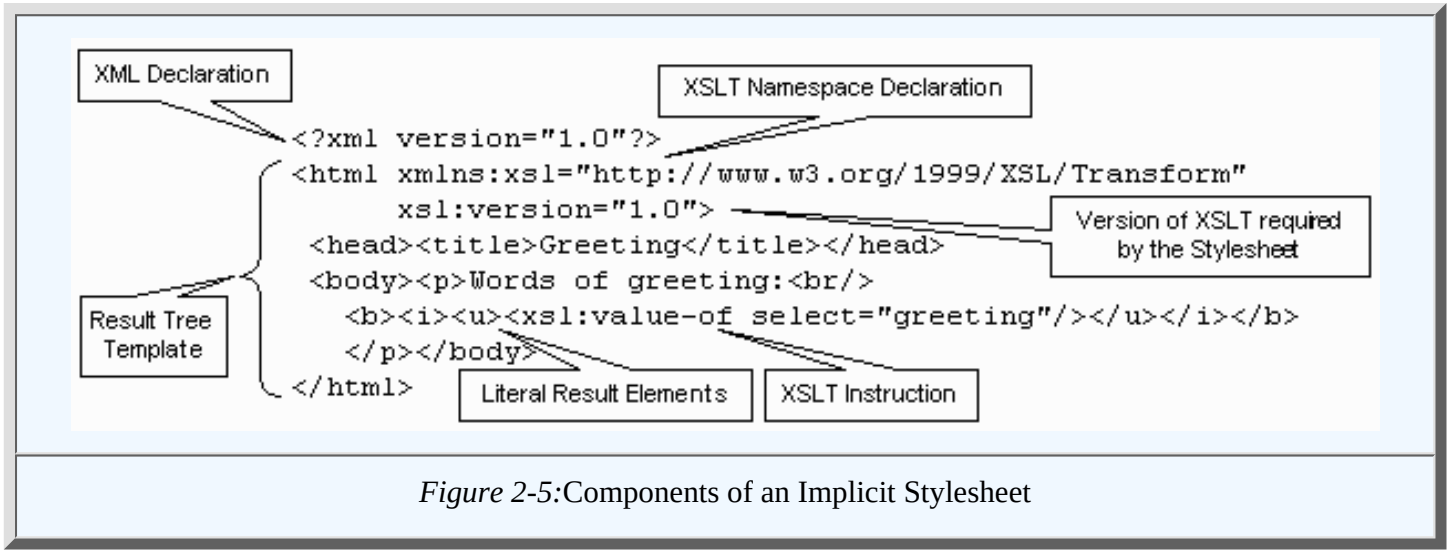


*Figure 2-5:*Components of an Implicit Stylesheet

The XML declaration is consumed by the XML processor embedded within the XSLT processor, thus the XSLT processor never sees it. The remainder of the file is considered the result tree template for an implicit rule for the root of the document, describing the shape of the entire outcome of the transformation.

The document element is named "`html`" and contains the namespace and version declarations of the XSLT language. Any element type within the result tree template that is qualified by the prefix assigned to the XSLT namespace URI is recognized as an XSLT instruction. No extension instruction namespaces are declared, thus all other element types in the instance are literal result elements. Indeed, the document element is a literal result element as it, too, is not an instruction.

2.2.3 Stylesheet requirements

Every XSLT stylesheet must identify the namespace prefix used therein for XSLT instructions. The default namespace cannot be used for this purpose. The namespace URI associated with the prefix must be the value `http://www.w3.org/1999/XSL/Transform` . It is a common practice to use the prefix `xsl` to identify the XSLT vocabulary, though this is only convention and any valid prefix can be used.

XSLT processor extensions are outside the scope of the XSLT vocabulary, so other URI values must be used to identify extensions.

The stylesheet must also declare the version of XSLT required by the instructions used therein. The attribute is named `version` and must accompany the namespace declaration in the wrapper element instruction as `version="`*`version-number`*`"` . In an implicit stylesheet where the XSLT namespace is declared in an element that is not an XSLT instruction, the namespace-qualified attribute declaration must be used as *`prefix`*`:version="`*`version-number`*`"` .

The version number is a numeric floating-point value representing the latest version of XSLT defining the

instructions used in the stylesheet. It need not declare the most capable version supported by the XSLT processor.

2.2.4 Instructions and literal result elements

XSLT instructions are only detected in the stylesheet tree and are not detected in the source tree. Instructions are specified using the namespace prefix associated with the XSLT namespace URI. The XSLT Recommendation describes the behavior of the XSLT processor for each of the instructions defined based on the instruction's element type (name).

Top-level instructions are considered and/or executed by the XSLT processor before processing begins on the source information. For better performance reasons, a processor may choose to not consider a top-level instruction until there is need within the stylesheet to use it. All other instructions are found somewhere in a result tree template and are not executed until that point at which the processor is asked to add the instruction to the result tree. Instructions themselves are never added to the result tree.

Some XSLT instructions are control constructs used by the processor to manage our stylesheets. The wrapper and top-level elements declare our globally scoped constructs. Procedural and process-control constructs give us the ability to selectively add only portions of templates to the result, rather than always adding an entire template. Logically-oriented constructs give us facilities to share the use of values and declarations within our own stylesheet files. Physically-oriented constructs give us the power to share entire stylesheet fragments.

Other XSLT instructions are result tree value placeholders. We declare how a value is calculated by the processor, or obtained from a source tree, or both calculated by the processor from a value from a source tree. The value calculation is triggered when the XSLT processor is about to add the instruction to the result tree. The outcome of the calculation (which may be nothing) is added to the result tree.

All other instructions engage customized non-standard behaviors and are specified using extension elements in a standardized fashion. These elements use namespace prefixes declared by our stylesheets to be instruction prefixes. Extension instructions may be either control constructs or result tree value placeholders.

Consider the simple example in our stylesheets used earlier in this chapter where the following instruction is used:

```
01  <xsl:value-of select="greeting"/>
```

Example 2-9: Simple value-calculation instruction in Example 2-4

This instruction uses the `select=` attribute to specify the XPath expression of some value to be calculated and added to the result tree. When the expression is a location in the source tree, as is this example, the value returned is the value of the first location identified using the criteria. When that location is an element, the value returned is the concatenation of all of the #PCDATA text contained therein.

This example instruction is executed in the context of the root of the source document being the focus. The child of the root of the document is the document element. The expression requests the value of the child named "`greeting` " of the root of the document, hence, the value of the document element named "`greeting` ". For

any source document where "`greeting`" is not the document element, the value returned is the empty string. For any source document where it is the document element, as is our example, the value returned is the concatenation of all #PCDATA text in the entire instance.

A literal result element is any element in a stylesheet that is not a top-level element and is not either an XSLT instruction or an extension instruction. A literal result element can use the default namespace or any namespace not declared in the stylesheet to be an instruction namespace.

When the XSLT processor reads the stylesheet and creates the abstract nodes in the stylesheet tree, those nodes that are literal result elements represent the nodes that are added to the result tree. Though the definition of those nodes is dictated by the XML syntax in the stylesheet entity, the syntax used does not necessarily represent the syntax that is serialized from the result tree nodes created from the stylesheet nodes.

Literal result elements marked up in the stylesheet entity may have attributes that are targeted for the XML processor used by the XSLT processor, targeted for the XSLT processor, or targeted for use in the result tree. Some attributes are consumed and acted upon as the stylesheet file is processed to build the stylesheet tree, while the others remain in the stylesheet tree for later use. Those literal result attributes remaining in the stylesheet tree that are qualified with an instruction namespace are acted on when they are asked to be added to the result tree.

2.2.5 Templates and template rules

Many XSLT instructions are container elements. The collection of literal result elements and other instructions being contained therein comprises the XSLT template for that instruction. A template can contain only literal result elements, only instruction elements, or a mixture of both. The behavior of the stylesheet can ask that a template be added to the result tree, at which point the nodes for literal result elements are added and the nodes for instructions are executed.

Consider again the simple example in our stylesheets used earlier in this chapter where the following template is used:

```
01  <b><i><u><xsl:value-of select="greeting"/></u></i></b>
```

Example 2-10: Simple template in Example 2-4

This template contains a mixture of literal result elements and an instruction element. When the XSLT processor adds this template to the result tree, the nodes for the `<b>` , `<i>` and `<u>` elements are simply added to the tree, while the node for the `xsl:value-of` instruction triggers the processor to add the outcome of instruction execution to the tree.

A template rule is a declaration to the XSLT processor of a template to be added to the result tree when certain conditions are met by source locations visited by the processor. Template rules are either top-level elements explicitly written in the stylesheet or built-in templates assumed by the processor and implicitly available in all stylesheets.

The criteria for adding a written template rule's template to the result tree are specified in a number of attributes, one of which must be the `match=` attribute. This attribute is an XPath pattern expression, which is a subset of XPath expressions in general. The pattern expression describes preconditions of source tree nodes. The stylesheet writer is responsible for writing the preconditions and other attribute values in such a way as to unambiguously provide a single written or built-in template for each of the anticipated source tree conditions.

In an implicitly declared stylesheet, the entire file is considered the template for the template rule for the root of the document. This template rule overrides the built-in rule implicitly available in the XSLT processor.

Back to the simple example in our explicitly declared stylesheet used earlier in this chapter, the following template rule is declared:

```
01  <xsl:template match="/">
02      <b><i><u><xsl:value-of select="greeting"/></u></i></b>
03  </xsl:template>
```

Example 2-11: Simple template rule in Example 2-4

This template rule defines the template to be added to the result tree when the root of the document is visited. This written rule overrides the built-in rule implicitly available in the XSLT processor. The template is the same template we were discussing earlier: a set of result tree nodes and an instruction.

The XSLT processor begins processing by visiting the root of the document. This gives control to the stylesheet writer. Either the supplied template rule or built-in template rule for the root of the document is processed, based on what the writer has declared in the stylesheet. The writer is in complete control at this early stage and all XSLT processor behavior is dictated what the writer asks to be calculated and where the writer asks the XSLT processor to visit.

2.2.6 Approaches to stylesheet design

The last discussion in this two-chapter introduction regards how to approach using templates and instructions when writing a stylesheet. Two distinct approaches can be characterized. Choosing which approach to use when depends on your own preferences, the nature of the source information, and the nature of the desired result.

Note:	I refer to these two approaches as either stylesheet-driven or data-driven, though the former might be misconstrued. Of course all results are stylesheet-driven because the stylesheet dictates what to do, so the use of the term involves some nuance. By *stylesheet-driven* I mean that the order of the result is a result of the stylesheet tree having explicitly instructed the adding of information to the result tree. By *data-driven* I mean that the order of the result is a result of the source tree ordering having dictated the adding of information to the result tree.

2.2.6.1 Pulling the input data

When the stylesheet writer knows the location of and order of data found in the source tree, and the writer wants to add to the result a value from or collection of that data, then information can be pulled from the source tree on demand. Two instructions are provided for this purpose: one for obtaining or calculating a single string value to add to the result; and one for adding rich markup to the result based on obtaining as many values as may exist in the tree.

The writer uses the `<xsl:value-of select="XPath-expression"/>` instruction in a stylesheet's element content to calculate a single value to be added to the result tree. The instruction is always empty and therefore does not contain a template. This value calculated can be the result of function execution, the value of a variable, or the value of a node selected from the source tree. When used in the template of various XSLT instructions the outcome becomes part of the value of a result element, attribute, comment, or processing instruction.

Note there is also a shorthand notation called an "attribute value template" that allows the equivalent to `<xsl:value-of>` to be used in a stylesheet's attribute content.

To iterate over locations in the source tree, the `<xsl:for-each select="XPath-node-set-expression">` instruction defines a template to be processed for each instance, possibly repeated, of the selected locations. This template can contain literal result elements or any instruction to be executed. When processing the given template, the focus of the processor's view of the source tree shifts to the location being visited, thus providing for relative addressing while moving through the information.

These instructions give the writer control over the order of information in the result. The data is being pulled from the source on demand and added to the result tree in the stylesheet-determined order. When collections of nodes are iterated, the nodes are visited in document order. This implements a stylesheet-driven approach to creating the result.

An implicitly-declared stylesheet is obliged to use only these "pull" instructions and must dictate the order of the result with the above instructions in the lone template.

2.2.6.2 Pushing the input data

The stylesheet writer may not know the order of the data found in the source tree, or may want to have the source tree dictate the ordering of content of the result tree. In these situations, the writer instructs the XSLT processor to visit source tree nodes and to apply to the result the templates associated with the nodes that are visited.

The `<xsl:apply-templates select="XPath-node-expression">` instruction visits the source tree nodes described by the node expression in the `select=` attribute. The writer can choose any relative, absolute, or arbitrary location or locations to be visited.

Each node visited is pushed through the stylesheet to be caught by template rules. Template rules specify the template to be processed and added to the result tree. The template added is dictated by the template rule matched for the node being pushed, not by a template supplied by the instruction when a node is being pulled. This distinguishes the behavior as being a data-driven approach to creating the result, in that the source determines the ultimate order of the result.

An implicitly-declared stylesheet can only push information through built-in template rules, which is of limited value. As well, the built-in rules can be mimicked entirely by using pull constructs, thus they need never be used. There is no room in the stylesheet to declare template rules in an implicitly-declared stylesheet since there is no wrapper stylesheet instruction.

An explicitly-declared stylesheet can either push or pull information because there is room in the stylesheet to define the top-level elements, including any number of template rules required for the transformation.

Putting it all together

We are *not* obliged to use only one approach when we write our stylesheets. It is very appropriate to push where the order is dictated by the source information and to pull when responding to a push where the order is known by the stylesheet. The most common use of this combination in a template is localized pull access to values that are relative to the focus being matched by nodes being pushed.

Note that push-oriented stylesheets more easily accommodate changes to the data and are more easily exploited by others who wish to reuse the stylesheets we write. The more granularity we have in our template rules, the more flexibly our stylesheets can respond to changes in the order of data. The more we pull data from our source tree, the more dependent we are on how we have coded the access to the information. The more we push data through our stylesheet, the less that changes in our data impact our stylesheet code.

Look again at the examples discussed earlier in this article and analyze the use of the above pull and push constructs to meet the objectives of the transformations.

These introductions and samples in this article have set the context, and only scratch the surface of the power of XSLT to effect the transformations we need when working with our structured information.

XML.com has continuing coverage and tutorials about XPath and XSLT in its regular column, Transforming XML.

This is a prose version of an excerpt from the book "Practical Transformation Using XSLT and XPath" (Eighth Edition ISBN 1-894049-05-5 at the time of this writing) published by Crane Softwrights Ltd., written by G. Ken Holman; this excerpt was edited by Stan Swaren, and reviewed by Dave Pawson.

Click Here
Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate...
THAWTE
www.thawte.com

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Pre-conference workshops: 25th July
Conference: 26th&27th July
Holiday Inn London, Victoria
Register Now

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter
search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

COLUMNS
XML-Deviant

What is XLink?

by Fabio Arciniegas A.
September 18, 2000

> "Only connect! That was the whole of the sermon"
>
> -- *E. M. Forster (1879 - 1970)*

Table of Contents

- Introduction
- An Example XLink
- XLink Reference
  - The XLink Type Attribute
  - XLink Types: Use and Composition
  - Simple Links
- Tools and References
- Conclusion

The very nature of the success of the Web lies in its capability for linking resources. However, the unidirectional, simple linking structures of the Web today are not enough for the growing needs of an XML world. The official W3C solution for linking in XML is called XLink (*XML Linking Language*). This article explains its structure and use according to the most recent Candidate Recommendation (July 3, 2000).

Overview

Every developer is familiar with the linking capabilities of the Web today. However, as the use of XML grows, we quickly realize that simple tags like `<A HREF="elem_lessons.html">Freud</A>` are not going to be enough for many of our needs.

Consider, for example the problem of creating an XML-based help system similar to ones used in some PC applications. Among other things (such as displaying amusingly animated characters), the system might be capable of performing the following actions when a user clicks on a topic:

- Opening an explanatory text (with a link back to the main index)

Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES

XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX

Syntax Checker
XML Testbed

- Opening a window and simulate the actions to be taken (e.g., going to the "Edit" menu and pressing "Include Image")
- Opening up a relevant dialog (e.g, a file chooser for the image to include)

Trying to code something like this (links with multiple targets, directions, and roles) in XML while having old "<a href..." in mind is confusing, and leads people to questions like the following:

- What is the "correct" tag for links in XML?>
- If there is such a magic element, how can I make it point to more than one resource?
- What if I want links to have different meanings relevant to my data? E.g., the "`motherhood`" and "`friendship`" relationships between two "`person`" elements

In answer to these and many other linking questions, this article describes the structure and use of XLink. The article is composed of three parts: a brief example that illustrates the basics of the language, a complete review of the structure of XLink, and a list of XLink-related resources. The resources include some XSLT transformations that enable your HTML output to simulate required XLink behavior on today's browsers.

Pages: **1**, 2, 3

Click Here
Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate...
THAWTE
www.thawte.com

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Pre-conference workshops: 25th July
Conference: 26th&27th July
Holiday Inn London, Victoria
Register Now

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

SSH Tunneling Secures 802.11
In the Wireless DevCenter

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

What is XLink?
by Fabio Arciniegas A. | Pages: 1, **2**, 3

Print This Article

Table of Contents

•Introduction
•An Example XLink
•XLink Reference
 •The XLink Type Attribute
 •XLink Types: Use and Composition
 •Simple Links
•Tools and References
•Conclusion

Before we start to dissect the structure of XLink, let's examine a concrete example.

The Artist/Influence problem

Suppose you want to express in XML the relationship between artists and their environment. This includes making links from an artist to his/her influences, as well as links to descriptions of historical events of their time. The data for each artist might be written in a file like the following:

```
<?xml version="1.0"?>
<artistinfo>
 <surname>Modigliani</surname>
 <name>Amadeo</name>
 <born>July 12, 1884</born><died>January 24, 1920</died>
 <biography>
   <p>In 1906, Modigliani settled in Paris, where ...</p>
 </biography>
</artistinfo>
```

Also, brief descriptions of time periods are included in separate files such as:

```
<?xml version="1.0"?>
<period>
 <city>Paris</city>
 <country>France<country>
 <timeframe begin="1900" end="1920"/>
 <title>Paris in the early 20th century (up to the twenties)</title>
 <end>Amadeo</end>
 <description>
   <p>During this period, Russian, Italian, ...</p>
 </description>
</period>
```

Fulfilling our requirement (i.e. creating a file that relates artists to their influences and periods) is a task beyond a simple strategy like adding "a" or "img" links to the above documents, for several reasons:

- A single artist has many influences (a link points from one resource to many).
- A single artist has associations with many periods.
- The link itself must be semantically meaningful. (Having an influence is not the same as belonging to a period, and we want to express that in our document!)

The XLink Solution

In XLink we have two type of linking elements: *simple* (like "a" and "img" in HTML) and *extended*. Links are represented as elements. However, XLink does not impose any particular "correct" name for your links; instead, it lets you decide which elements of your own are going to serve as links, by means of the XLink attribute *type*. An example snippet will make this clearer:

```
<environment xlink:type="extended">
    <!-- This is an extended link -->
    <!-- The resources involved must be included/referenced here -->
</environment>
```

Now that we have our extended link, we must specify the resources involved. Since the artist and movement information are stored outside our own document (so we have no control over them), we use XLink's *locator* elements to reference them. Again, the strategy is not to impose a tag name, but to let you mark your elements as locators using XLink attributes:

```
<environment xmlns:xlink="http://www.w3.org/1999/xlink"
    xlink:type="extended">
```

XML
Products
Tools
Training
O'REILLY xml.com

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

```
        <!-- The resources involved in our link are the artist -->
        <!-- himself, his influences and the historical references -->
        <artist    xlink:type="locator" xlink:label="artist"
              xlink:href="modigliani.xml"/>
        <influence xlink:type="locator" xlink:label="inspiration"
              xlink:href="cezanne.xml"/>
        <influence xlink:type="locator" xlink:label="inspiration"
              xlink:href="lautrec.xml"/>
        <influence xlink:type="locator" xlink:label="inspiration"
              xlink:href="rouault.xml"/>
        <history   xlink:type="locator" xlink:label="period"
              xlink:href="paris.xml"/>
        <history   xlink:type="locator" xlink:label="period"
              xlink:href="kisling.xml"/>
    </environment>
```

Only one thing is missing: We must specify how the resources relate to each other. We do this by specifying *arcs* between them:

```
    <environment xmlns:xlink="http://www.w3.org/1999/xlink"
         xlink:type="extended">
         <!-- an artist is bound to his influences and history -->
         <artist    xlink:type="locator" xlink:role="artist"
               xlink:href="modigliani.xml"/>
         <influence xlink:type="locator" xlink:label="inspiration"
               xlink:href="cezanne.xml"/>
         <influence xlink:type="locator" xlink:label="inspiration"
               xlink:href="lautrec.xml"/>
         <influence xlink:type="locator" xlink:label="inspiration"
               xlink:href="rouault.xml"/>
         <history   xlink:type="locator" xlink:label="period"
               xlink:href="paris.xml"/>
         <history   xlink:type="locator" xlink:label="period"
               xlink:href="kisling.xml"/>
         <bind xlink:type="arc" xlink:from="artist"
               xlink:to="inspiration"/>
         <bind xlink:type="arc" xlink:from="artist"
               xlink:to="period"/>
       </environment>
```

As you can see, using XLink, our problem is reduced to creating an XML file full of elements like the above, where all the resources and their relationships are clearly and elegantly specified.

In this section we saw a small example of the use and syntax of XLink. In the next one, we will examine in detail the constructs and rules of this linking mechanism.

Pages: 1, **2**, 3

Next Page▶

Click Here Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate... THAWTE www.thawte.com

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

What is XLink?
by Fabio Arciniegas A. | Pages: 1, 2, **3**

Print This Article

XLink Reference

Now that we have a basic idea of how XLink looks, it's time to dive into the details. This section presents all the constructs and rules contained in the XLink specification.

Table of Contents

•Introduction
•An Example XLink
•XLink Reference
  •The XLink Type Attribute
  •XLink Types: Use and Composition
  •Simple Links
•Tools and References
•Conclusion

Basics

XLink works by proving you with global attributes you can use to mark your elements as *linking elements*. In order to use linking elements, the declaration of the XLink namespace is required:

```
<my_element xmlns:xlink="http://www.w3.org/1999/xlink> ...
```

Using the global attributes provided by XLink, one may specify whether a particular element is a linking element, and many properties about it (e.g., when to load the linked resources, how to see them once they are loaded, etc.). The g*lobal attributes* provided by XLink are the following:

Type definition attribute	type
Locator attribute	href
Semantic attributes	role, arcrole, title
Behavior attributes	show, actuate
Traversal attributes	label, from, to

The next sections explain each of these attributes, their possible values and the rules that govern their use.

The XLink *type* attribute

The *type* attribute may have one of the following values:

- *simple*: a simple link
- *extended*: an extended, possibly multi-resource, link
- *locator*: a pointer to an external resource
- *resource*: an internal resource
- *arc*: a traversal rule between resources
- *title*: a descriptive title for another linking element

By convention, when an attribute includes the type attribute with a value V, we will refer to it as a *V-type element*, no matter what its actual name is.

```
<!-- bookref is a locator-type element -->
<bookref xlink:type="locator" ...
```

Two restrictions stem from the fact that an element belongs to a certain XLink type:

1. Given an element of a particular type, only elements of certain types are relevant as XLink subelements.

```
<!-- since A is a simple-type element, all the information
it needs is on the href attribute. It would make no
sense to have a locator-type subelement -->

<a xlink:type="simple" href="monet.html"> ... no other
xlink element would make sense here... </a>
```

2. Given an element of a particular type, only some XLink attributes apply:

```
<!-- since bookref is a locator-type element, it needs an href
attribute to point to the external resource, but it
```

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

```
would make no sense for it to have a from attribute, which
is reserved for arcs. -->
<bookref xlink:type="locator" href="ficciones.xml"/>
```

The following two tables summarize the attribute and subelement restrictions of each type (they are included here as a reference, but each element will be properly explained later on). In Table 1, "R" indicates "required," and "O" indicates "optional." A blank space indicates an invalid combination. Table 2 shows which XLink elements are permitted which XLink subelements.

Attribute	**simple**	**extended**	**locator**	**arc**	**resource**	**title**
type	R	R	R	R	R	R
href	O		R			
role	O	O	O		O	
arcrole	O			O		
title	O	O	O	O	O	
show	O			O		
actuate	O			O		
label			O		O	
from				O		
to				O		

Table 1 - Attribute usage (from the W3C specification)

Parent type	**Significant child element types**
simple	-
extended	locator, arc, resource, title
locator	title
arc	title
resource	-
title	-

Table 2 - Significant child types (from the W3C specification)

XLink Types: Use and Composition

Let's review each of the XLink types. To do this, we'll use an example of linking actresses and the movies they played in.

Resources (*resource-type* and *locator-type* elements)

The resources involved in a link can be either local (resource-type elements) or remote (pointed to by locator-type elements). For a rough equivalent in HTML, think of resource-type elements as "<a name..>" and locator-type elements as "<a href...>". The following code shows a DTD declaration of a resource element:

```
<!ELEMENT actress   (first_name,surname)>
<!ATTLIST actress
        xlink:type        (resource)       #FIXED "resource"
        xlink:title       CDATA            #IMPLIED
        xlink:label       NMTOKEN          #IMPLIED>
        xlink:role        CDATA           #IMPLIED
```

Note that the element has another two XLink-based attributes besides *xlink:type*. The first one, "title," is a *semantic attribute* used to give a short description of the resource. The second one, "label," is a *traversal attribute*, used to identify the element later, when we build arcs. The third attribute, "role," is used for describing a property of the resource.

An actress element may look like the following:

```
<actress xlink:label="maria">
   <first_name>Brigitte</first_name>
   <surname>Helm</surname>
</actress>
```

It is important to note also that the subelements of resource-type elements (here, the *first_name* and *surname* elements) have no significance for XLink (see Table 2).

As we mentioned before, remote resources are pointed to by locators. Here is the DTD for a locator-type element:

```
<!ELEMENT movie             EMPTY>
```

```
<!ATTLIST movie
        xlink:type        (locator)        #FIXED "locator"
        xlink:title       CDATA            #IMPLIED
        xlink:role      CDATA           #IMPLIED
        xlink:label       NMTOKEN          #IMPLIED
        xlink:href        CDATA            #REQUIRED>
```

Locators can have the same attributes as resources (i.e., *title*, *label*, and *role*), plus a required *href* semantic attribute, which points to the remote resource. A locator movie element will look like the following:

```
<movie xlink:label="metropolis" xlink:href="metropolis.xml"/>
```

Navigation rules (arc-type elements)

The relationships between resources involved in a link are specified using arcs. Arc-type elements (i.e. those with xlink:type="arc") use the "to" and "from" attributes to designate the start and end points of an arc:

```
<acted xlink:type="arc" xlink:from="maria" xlink:to="metropolis"/>
```

Aside from the *traversal attributes* "to" and "from," arcs may include the following:

- *show*: This attribute is used to determine the desired presentation of the ending resource. Its possible values are "new" (open a new window), "replace" (load the referenced resource in the same window), "embed" (embed the pointed resource -- a movie, for example), "none" (unrestricted), and "other" (unrestricted by the XLink spec, but the processor should look into the subelements for further information).
- *title*: Just as with resources, this is simply a human-readable string with a short description for the arc.
- *actuate*: This attribute is used to determine the timing of traversal to the ending resource. Its possible values are "onLoad" (load the ending resource as soon as the start resource is found), "onRequest" (e.g., user clicks the link), "other," and "none."
- *arcrole*: The advanced uses of arcrole (and its counterpart, the role attribute) are beyond the scope of this article. (Please refer to section 5 of the XLink specification for a discussion on linkbases). For our discussion, suffice it to say that this attribute must be a URI reference for some description of the arc role.

Note that XLinks permit both inbound and outbound links. Outbound links are akin to normal HTML links, where a link is made from the current document to an external resource. An inbound link is constituted by an arc from an external resource, located with a locator-type element, into an internal resource.

The following DTD will illustrate the above attributes:

```
<!ELEMENT acted EMPTY>
<!ATTLIST acted
        xlink:type       (arc)           #FIXED "arc"
        xlink:title      CDATA           #IMPLIED
        xlink:show       (new | replace |
                          embed | other | none)
                                         #IMPLIED
        xlink:from       NMTOKEN         #IMPLIED
        xlink:to         NMTOKEN         #IMPLIED>
```

Putting together our resource and locator examples with this arc, we have the following snippet of an XML instance:

```
<!-- A local resource -->
<actress xlink:label="maria">
    <first_name>Brigitte</first_name>
    <surname>Helm</surname>
</actress>

<!-- A remote resource -->
<movie xlink:label="metropolis" xlink:href="metropolis.xml"/>

<!-- An arc that binds them -->
<acted xlink:type="arc" xlink:from="maria" xlink:to="metropolis"/>
```

In order to encapsulate relationships like the above we need containers, that is, *extended-type* XLink elements

Extended links (*extended-type* elements)

Extended links are marked by the type "extended" and may contain locators (pointing to remote resources), local resources, arcs, and a title. The diagram below illustrates the composition of an extended link.

One can simply consider the extended-link elements as meaningful wrappers that provide a nest for resources and arcs:

```
<!ELEMENT divas (actress,movie,acted)*>
<!ATTLIST divas
        xmlns:xlink  CDATA      #FIXED "http://www.w3.org/1999/xlink"
        xlink:type   (extended) #FIXED "extended"
        xlink:title  CDATA      #IMPLIED>
```

Putting together all the previous elements, we finally have a complete and valid extended link. (Note in particular the one-to-many link that has been generated, something previously not possible in HTML.)

```
    <divas xlink:title="German divas 1920s">
        <actress xlink:label="maria">
          <first_name>Brigitte</first_name>
          <surname>Helm</surname>
        </actress>

        <movie xlink:label="silent" xlink:title="Metropolis"
               xlink:href="metropolis.xml"/>
        <movie xlink:label="silent" xlink:title="Alaraune"
               xlink:href="alaraune.xml"/>
        <acted xlink:type="arc" xlink:from="maria" xlink:to="silent"/>
        ...
     <divas>
```

Title elements

An alternative way to provide titles to extended, locator, and arc type elements is by using a *title-type* subelement (xlink:type="title"). This was included in order to have a standard way for applications to express complex titles that include more than a string. (For instance, one might use multiple titles in different languages, to provide localization features.) The contents of title-type elements are not constrained by XLink.

Simple links

Simple links are, conceptually, a subset of extended links. They exist as a notation for links where you don't need the overhead of an entire extended link. All the XLink-related aspects of a simple link are encapsulated on one element (i.e., XLink doesn't care about the subelements of a simple link).

The valid XLink attributes of a simple link are "href" (just like in HTML's "a" or "img"), "title," "role," "arcrole," "show," and "actuate," which keep the same semantics as when used in arc-type elements.

The following shows a typical simple link element:

```
<!-- first, a DTD declaration -->
<!ELEMENT director (#PCDATA)>
<!ATTLIST director
    xmlns:xlink    CDATA       #FIXED "http://www.w3.org/1999/xlink"
    xlink:type     (simple)    #FIXED "simple"
    xlink:href     CDATA       #IMPLIED
    xlink:show     (new)       #FIXED "new"
```

```
    xlink:actuate  (onRequest) #FIXED "onRequest">

          ...
<!-- now, a typical instance -->
<director xlink:href="fincher.xml">David Fincher</director>
```

That's all there is to it. We have covered all the types and attributes of XLink. As you can see, this is a powerful but compact specification that is bound to prove useful in future projects. We will wrap up by presenting some pointers to useful XLink tools.

Tools and references

The following is a (non-exhaustive) list of XLink-aware tools and references you might find useful for your projects:

1. Mozilla M17 Browser (Mozilla). Open source browser with restricted XLink support
2. Link (Justin Ludwig). A small, XLink-aware XML browser
3. psgml-xpointer.el (David Megginson). A very useful extension to psgml for emacs that generates XPointer expressions
4. *Reusable XLink XSLT transformations* (Fabio Arciniegas A.). This set of XSLT templates allow the transformation of extended links to HTML and JavaScript representations.
5. The XLink Specification (W3C - July 3, 2000)
6. XMLhack XLink news Latest XLink news and software releases.

Conclusion

XLink is a powerful and compact specification for the use of links in XML documents. This article has explored the structure and basic uses of XLink as described in the current W3C spec (July 3rd, 2000).

Even though XLink has not been implemented in any of the major commercial browsers yet, its impact will be crucial for the XML applications of the near future. Its extensible and easy-to-learn design should prove an advantage as the new generation of XML applications develop. For questions and comments, please contact the author.

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site Click here!

COLUMNS
XML-Deviant

What is XLink?

by Fabio Arciniegas A.
September 18, 2000

> "Only connect! That was the whole of the sermon"
>
> -- *E. M. Forster (1879 - 1970)*

Table of Contents

•Introduction
•An Example XLink
•XLink Reference
  •The XLink Type Attribute
  •XLink Types: Use and Composition
  •Simple Links
•Tools and References
•Conclusion

The very nature of the success of the Web lies in its capability for linking resources. However, the unidirectional, simple linking structures of the Web today are not enough for the growing needs of an XML world. The official W3C solution for linking in XML is called XLink (*XML Linking Language*). This article explains its structure and use according to the most recent Candidate Recommendation (July 3, 2000).

Overview

Every developer is familiar with the linking capabilities of the Web today. However, as the use of XML grows, we quickly realize that simple tags like `<A HREF="elem_lessons.html">Freud</A>` are not going to be enough for many of our needs.

Consider, for example the problem of creating an XML-based help system similar to ones used in some PC applications. Among other things (such as displaying amusingly animated characters), the system might be capable of performing the following actions when a user clicks on a topic:

- Opening an explanatory text (with a link back to the main index)

Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

- Opening a window and simulate the actions to be taken (e.g., going to the "Edit" menu and pressing "Include Image")
- Opening up a relevant dialog (e.g, a file chooser for the image to include)

Trying to code something like this (links with multiple targets, directions, and roles) in XML while having old "<a href..." in mind is confusing, and leads people to questions like the following:

- What is the "correct" tag for links in XML?>
- If there is such a magic element, how can I make it point to more than one resource?
- What if I want links to have different meanings relevant to my data? E.g., the "`motherhood`" and "`friendship`" relationships between two "`person`" elements

In answer to these and many other linking questions, this article describes the structure and use of XLink. The article is composed of three parts: a brief example that illustrates the basics of the language, a complete review of the structure of XLink, and a list of XLink-related resources. The resources include some XSLT transformations that enable your HTML output to simulate required XLink behavior on today's browsers.

Pages: **1**, 2, 3

Next Page

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

O'REILLY OPEN SOURCE CONVENTION
July 23–27, 2001
San Diego
conferences.oreilly.com

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter
search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant

What is RDF?

by Tim Bray
January 24, 2001

This article was first published as "RDF and Metadata" on XML.com in June 1998. It has been updated by ILRT's Dan Brickley, chair of the W3C's RDF Interest Group, to reflect the growing use of RDF and updates to the specification since 1998.

Table of Contents

•The Right Way to Find Things
•It's All Different Behind the Scenes
•Not Just For Searching
•What About the Web?
•Divine Metadata for the Web
•Introducing RDF
•Why Not Just Use XML?
•The Devil is in the Details
•Vocabularies
•What RDF Might Mean
•Getting started with RDF
•Developer Community

The Right Way to Find Things

RDF stands for Resource Description Framework. RDF is built for the Web, but let's leave the Web behind for now and think about how we find things in the real world.

Scenario 1: The Library

You're in a library to find books on raising donkeys as pets. In most libraries these days you'd use the computer lookup system, basically an electronic version of the old card file. This system allows you to list books by author, title, subject, and so on. The list includes the date, author, title, and lots of other useful information, including (most important of all) where each book is.

Scenario 2: The Video Store

Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

You're in a video store and you want a movie by John Huston. A large modern video store offers a lookup facility that's similar to the library's. Of course, the search properties are different (director, actors, and so on) but the results are more or less the same.

Scenario 3: The Phone Book

You're working late at a customer's office in South Denver, and it seems that a pizza is essential if work is to continue. Fortunately, every office comes equipped with a set of Yellow Pages that, when properly used, can lead to quick pizza delivery.

The Common Thread

What do all these situations have in common, and what differences lie behind the scenes? First of all, each of these systems is based on *metadata*, that is, information about information. In each case, you need a piece of information (the book's location, the video's name, the pizza joint's phone number) you don't have. In each case, you use metadata (information about information) to get it.

We're all used to this stuff; metadata ordinarily comes in named chunks (subject, director, business category) that associate lookup information ("donkeys", "John Huston", "Pizza, South Side") with the information you're really after.

Here's a subtle but important point -- in theory, metadata is not really necessary: you could go through the library one book at a time looking for donkey books, or through the video store shelves until you found your movie, or call all the numbers in your area code until you find pizza delivery. But that would be very wasteful, in fact, it would be stupid. Metadata is the way to go.

It's All Different Behind the Scenes

In each of our scenarios, we used metadata, and we used it in remarkably similar ways. Does this mean that the library, the video store, and the phone company all use the same metadata setup? Of course not. Every library has a choice among at least two systems for organizing their books, and among many vendors who will sell them software to do the looking-up. The same is obviously true for video stores and phone companies.

In fact most such products define their own system of metadata and their own facilities for storing and managing it. They typically do *not* offer facilities for sharing or interchanging it. This doesn't cause too much of a problem, assuming they do a decent job with the user interface. We are comfortable enough with the general process we call "looking things up" (really, searching via metadata) that we are able to adapt and use all these different systems.

Not Just For Searching

The most common daily use of metadata is to aid our discovery of things. But there are lots of other uses going on behind the scenes. The library and video store are storing other metadata that you don't see: how often the books and videos are being used, how much it cost to buy them, where to go for a replacement, etc. Running a library or a video store would be unthinkable without metadata. Similarly, the phone company, of course, uses its metadata, most obviously to *print* the Yellow Pages, but for many other internal management and administration tasks.

What About the Web?

The Web is a lot like a really *really* big library. There are millions of things out there, and if you know the URL (in effect a kind of call number) you can get them. Since the Web has books, movies, and pizza joints, the number of ways you might want to look things up includes all the things a library uses, plus all the things the video store uses, plus all the things the Yellow Pages use, and lots more.

The problem at the moment is that there is hardly any metadata on the Web. So how do we find things? Mostly by using dumb, brute force techniques. The dumb, brute force is supplied by the wandering web robots of search engine sites like Altavista, Infoseek, and Excite. These sites do the equivalent of going through the library, reading every book, and allowing us to look things up based on the words in the text. It's not surprising that people complain about search results, or that the robots are always way behind the growth and change of the Web.

In fact there is one metadata-based general purpose lookup facility: Yahoo! Yahoo doesn't use a robot. When you search through Yahoo, you're searching through human-generated subject categories and site labels. Compared to the amount of metadata that a library maintains for its books, Yahoo! is pitiful; but its popularity is clear evidence of the power of (even limited) metadata.

Pages: **1**, 2, 3

Two powerful platforms. One easy choice.
IBM.
THE BOOK OF @ BUSINESS
> Click to explore the AIX advantage.

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

O'REILLY OPEN SOURCE CONVENTION July 23-27, 2001 San Diego conferences.oreilly.com

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

What is RDF?
by Tim Bray | Pages: 1, **2**, 3

Print This Article

Divine Metadata for the Web

Table of Contents

•The Right Way to Find Things
•It's All Different Behind the Scenes
•Not Just For Searching
•What About the Web?
•Divine Metadata for the Web
•Introducing RDF
•Why Not Just Use XML?
•The Devil is in the Details
•Vocabularies
•What RDF Might Mean
•Getting started with RDF
•Developer Community

People who have thought about these problems, including many librarians and webmasters, generally agree that the Web urgently needs metadata. What would it look like? If the Web had an all-powerful Grand Organizing Directorate (at www.GOD.org), it would think up a set of lookup fields such as Author, Title, Date, Subject, and so on. The Directorate, being, after all, GOD, would simply decree that all Web pages start using this divine Metadata, and that would be that. Of course there would be some details such as how the Web sites ought to package up and interchange the metadata, and we all know that the Devil is in the details, but GOD can lick the Devil any day.

In fact, there is no www.GOD.org. For this reason, there is no chance that everyone will agree to start using the same metadata facilities. If libraries, which have existed for hundreds of years, can't agree on a single standard, there's not much chance that the Web will.

Does this mean that there is no chance for metadata? That everyone is going to have to build their own lookup keys and values and software, and that we're going to be stuck using dumb, brute force robots forever?

No. As we observed with our three search scenarios, metadata operations have an awful lot in common, even when the metadata is different. RDF is an effort to identify these common threads and provide a way for Web architects to use them to provide useful Web metadata without divine intervention.

Introducing RDF

Resource Description Framework, as its name implies, is a framework for describing and interchanging metadata. It is built on the following rules.

1. A **Resource** is anything that can have a URI; this includes all the Web's pages, as well as individual elements of an XML document. An example of a resource is a draft of the document you are now reading and its URL is `http://www.textuality.com/RDF/Why.html`
2. A **Property** is a Resource that has a name and can be used as a property, for example `Author` or `Title`. In many cases, all we really care about is the name; but a Property needs to be a resource so that it can have its own properties.
3. A **Statement** consists of the combination of a Resource, a Property, and a value. These parts are known as the 'subject', 'predicate' and 'object' of a Statement. An example Statement is "The Author of `http://www.textuality.com/RDF/Why.html` is Tim Bray." The value can just be a string, for example "Tim Bray" in the previous example, or it can be another resource, for example "The Home-Page of `http://www.textuality.com/RDF/Why.html` is `http://www.textuality.com`."
4. There is a straightforward method for expressing these abstract Properties in XML, for example:

```
<rdf:Description about='http://www.textuality.com/RDF/Why-RDF.html'>
<Author>Tim Bray</Author>
<Home-Page rdf:resource='http://www.textuality.com' />
</rdf:Description>
```

RDF is carefully designed to have the following characteristics.

Independence

Since a Property is a resource, any independent organization (or even person) can invent them. I can invent one called Author, and you can invent one called Director (which would only apply to resources that are associated with movies), and someone else can invent one called Restaurant-Category. This is necessary since we don't have a GOD to take care of it for us.

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

Interchange

Since RDF Statements can be converted into XML, they are easy for us to interchange. This would probably be necessary even if we *did* have a GOD.

Scalability

RDF statements are simple, three-part records (Resource, Property, value), so they are easy to handle and look things up by, even in large numbers. The Web is already big and getting bigger, and we are probably going to have (literally) billions of these floating around (millions even for a big Intranet). Scalability is important.

Properties are Resources

Properties can have their own properties and can be found and manipulated like any other Resource. This is important because there are going to be lots of them; too many to look at one by one. For example, I might want to know if anyone out there has defined a Property that describes the genre of a movie, with values like Comedy, Horror, Romance, and Thriller. I'll need metadata to help with that.

Values Can Be Resources

For example, most web pages will have a property named Home-Page which points at the home page of their site. So the values of properties, which obviously have to include things like title and author's name, also have to include Resources.

Statements Can Be Resources

Statements can also have properties. Since there's no GOD to provide useful assertions for all the resources, and since the Web is way too big for us to provide our own, we're going to need to do lookups based on other people's metadata (as we do today with Yahoo!). This means that we'll want, given any Statement such as "The Subject of this Page is Donkeys", to be able to ask "Who said so? And When?" One useful way to do this would be with metadata; so Statements will need to have Properties.

Why Not Just Use XML?

XML allows you to invent tags, which may contain both text data and other tags. XML has a built-in distinction between *element types*, for example the `IMG` element type in HTML, and *elements*, for example an individual `<img src='Madonna.jpg'>`; this corresponds naturally to the distinction between Properties and Statements. So it seems as though XML documents should be a natural vehicle for exchanging general purpose metadata.

XML, however, falls apart on the **Scalability** design goal. There are two problems:

1. The order in which elements appear in an XML document is significant and often very meaningful. This seems highly unnatural in the metadata world. Who cares whether a movie's Director or Title is listed first, as long as both are available for lookups? Furthermore, maintaining the correct order of millions of data items is expensive and difficult, in practice.
2. XML allows constructions like

```
<Description>The value of this property contains some
text, mixed up with child properties such as its temperature
(<Temp>48</Temp>) and longitude
(<Longt>101</Longt>). [&Disclaimer;]</Description>
```

When you represent general XML documents in computer memory, you get weird data structures that mix trees, graphs, and character strings. In general, these are hard to handle in even moderate amounts, let alone by the billion.

On the other hand, something like XML is an absolutely necessary part of the solution to RDF's **Interchange** design goal. XML is unequalled as an exchange format on the Web. But by itself, it doesn't provide what you need in a metadata framework.

Pages: 1, **2**, 3 Next Page

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

What is RDF?
by Tim Bray | Pages: 1, 2, **3** Print This Article

The Devil is in the Details

Table of Contents

•The Right Way to Find Things
•It's All Different Behind the Scenes
•Not Just For Searching
•What About the Web?
•Divine Metadata for the Web
•Introducing RDF
•Why Not Just Use XML?
•The Devil is in the Details
•Vocabularies
•What RDF Might Mean
•Getting started with RDF
•Developer Community

The four general rules given above define the central ideas of RDF. It turns out that it takes quite a lot of abstract terminology and XML syntax to define them precisely enough that people can write computer programs to process them. In particular, turning Statements into Resources is quite tricky. It also turns out that in a (very) few cases, you do need to order your properties, and this requires quite a bit of syntax.

This article doesn't explain all these details; there are a variety of excellent resources to be found at http://www.w3.org/RDF that are designed to do just that.

Vocabularies

RDF, as we've seen, provides a model for metadata, and a syntax so that independent parties can exchange it and use it. What it

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

doesn't provide though is any Properties of its own. RDF doesn't define Author or Title or Director or Business-Category. That would be a job for GOD, if there were one. Since there isn't, it's a job for everyone.

It seems unlikely that one Property standing by itself is apt to be very useful. It is expected that these will come in packages; for example, a set of basic bibliographic Properties like Author, Title, Date, and so on. Then a more elaborate set from OCLC and a competing one from the Library of Congress. These packages are called **Vocabularies**; it's easy to imagine Property vocabularies describing books, videos, pizza joints, fine wines, mutual funds, and many other species of Web wildlife.

What RDF Might Mean

The Web is too big for anyone person to stay on top of. In fact, it contains information about a huge number of subjects, and for most of those subjects (such as fine wines, home improvement, and cancer therapy), the Web has too much information for any one person to stay on top of and much of anything else .

This means that opinions, pointers, indexes, and anything that helps people discover things are going to be commodities of very high value. Nobody thinks that everyone will use the same vocabulary (nor should they), but with RDF we can have a marketplace in vocabularies. Anyone can invent them, advertise them, and sell them. The good (or best-marketed) ones will survive and prosper. Probably most niches of information will come to be dominated by a small number of vocabularies, the way that library catalogs are today.

And even among people who are sharing the use of metadata vocabularies, there's no need to share the same software. RDF makes it possible to use multiple pieces of software to process the same metadata, and to use a single piece of software to process (at least in part) many different metadata vocabularies.

With any luck, this should make the Web more like a library, or a video store, or a phone book, than it is today.

Getting started with RDF

Since RDF became a W3C Recommendation in February 1999, a number of tools have been created by developers working with RDF. For an in-depth treatment of these, consult the W3C RDF home page. A number of other listings are available, including XML.com, XMLhack and Dave Beckett's RDF Resource Guide.

Developer Community

The main email list for RDF developer discussion is W3C's RDF Interest Group. A number of other RDF-related discussion lists exist, including the Mozilla-RDF forum (the Mozilla and Netscape 6 browsers make heavy use of RDF). More recently, the RDF-Logic list has been announced, providing a forum for the discussion of formal, logic-based approaches to knowledge representation for the Web. DARPA's DAML (DARPA Agent Markup Language) initiative uses the RDF-Logic list for discussions and announcements.

The RDF developer community is rather diverse, which is reflected in the nature of online discussions on the RDF lists. While one strand of RDF development is concerned with highly formal topics (RDF-Logic, DAML and so on), others are busy deploying simpler, more pragmatic applications for Web-based content and metadata syndication. All these themes meet (sometimes productively, sometimes confusingly) on the RDF Interest Group list, but they also typically each have a dedicated email list. For example, the RSS-DEV group has produced the RDF Site Summary (RSS) 1.0 Specification, which provides an RDF-based channel format, designed for interoperability with high level vocabularies such as Dublin Core as well as a variety of more application-specific RDF vocabularies.

Notes on Update (Dan Brickley)

This update to the 1998 article serves only to synchronize it with recent RDF terminology. Since this document was first published, the W3C has published the Model and Syntax

specification as a Recommendation.

I have updated the markup example to use current RDF 1.0 syntax. There have also been some terminology changes: 'PropertyType' became 'Property', 'Property' became 'Statement'. I have also added a brief mention of subject/predicate/object terminology, and lowercased a few mentions 'Value' (since rdf:object replaced rdf:value for talking about the object of a statement).

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Pre-conference workshops: 25th July
Conference: 26th&27th July
Holiday Inn London, Victoria
Register Now

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant

What is RDF?

by Tim Bray
January 24, 2001

This article was first published as "RDF and Metadata" on XML.com in June 1998. It has been updated by ILRT's Dan Brickley, chair of the W3C's RDF Interest Group, to reflect the growing use of RDF and updates to the specification since 1998.

Table of Contents

•The Right Way to Find Things
•It's All Different Behind the Scenes
•Not Just For Searching
•What About the Web?
•Divine Metadata for the Web
•Introducing RDF
•Why Not Just Use XML?
•The Devil is in the Details
•Vocabularies
•What RDF Might Mean
•Getting started with RDF
•Developer Community

The Right Way to Find Things

RDF stands for Resource Description Framework. RDF is built for the Web, but let's leave the Web behind for now and think about how we find things in the real world.

Scenario 1: The Library

You're in a library to find books on raising donkeys as pets. In most libraries these days you'd use the computer lookup system, basically an electronic version of the old card file. This system allows you to list books by author, title, subject, and so on. The list includes the date, author, title, and lots of other useful information, including (most important of all) where each book is.

Scenario 2: The Video Store

Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

You're in a video store and you want a movie by John Huston. A large modern video store offers a lookup facility that's similar to the library's. Of course, the search properties are different (director, actors, and so on) but the results are more or less the same.

Scenario 3: The Phone Book

You're working late at a customer's office in South Denver, and it seems that a pizza is essential if work is to continue. Fortunately, every office comes equipped with a set of Yellow Pages that, when properly used, can lead to quick pizza delivery.

The Common Thread

What do all these situations have in common, and what differences lie behind the scenes? First of all, each of these systems is based on *metadata*, that is, information about information. In each case, you need a piece of information (the book's location, the video's name, the pizza joint's phone number) you don't have. In each case, you use metadata (information about information) to get it.

We're all used to this stuff; metadata ordinarily comes in named chunks (subject, director, business category) that associate lookup information ("donkeys", "John Huston", "Pizza, South Side") with the information you're really after.

Here's a subtle but important point -- in theory, metadata is not really necessary: you could go through the library one book at a time looking for donkey books, or through the video store shelves until you found your movie, or call all the numbers in your area code until you find pizza delivery. But that would be very wasteful, in fact, it would be stupid. Metadata is the way to go.

It's All Different Behind the Scenes

In each of our scenarios, we used metadata, and we used it in remarkably similar ways. Does this mean that the library, the video store, and the phone company all use the same metadata setup? Of course not. Every library has a choice among at least two systems for organizing their books, and among many vendors who will sell them software to do the looking-up. The same is obviously true for video stores and phone companies.

In fact most such products define their own system of metadata and their own facilities for storing and managing it. They typically do *not* offer facilities for sharing or interchanging it. This doesn't cause too much of a problem, assuming they do a decent job with the user interface. We are comfortable enough with the general process we call "looking things up" (really, searching via metadata) that we are able to adapt and use all these different systems.

Not Just For Searching

The most common daily use of metadata is to aid our discovery of things. But there are lots of other uses going on behind the scenes. The library and video store are storing other metadata that you don't see: how often the books and videos are being used, how much it cost to buy them, where to go for a replacement, etc. Running a library or a video store would be unthinkable without metadata. Similarly, the phone company, of course, uses its metadata, most obviously to *print* the Yellow Pages, but for many other internal management and administration tasks.

What About the Web?

The Web is a lot like a really *really* big library. There are millions of things out there, and if you know the URL (in effect a kind of call number) you can get them. Since the Web has books, movies, and pizza joints, the number of ways you might want to look things up includes all the things a library uses, plus all the things the video store uses, plus all the things the Yellow Pages use, and lots more.

The problem at the moment is that there is hardly any metadata on the Web. So how do we find things? Mostly by using dumb, brute force techniques. The dumb, brute force is supplied by the wandering web robots of search engine sites like Altavista, Infoseek, and Excite. These sites do the equivalent of going through the library, reading every book, and allowing us to look things up based on the words in the text. It's not surprising that people complain about search results, or that the robots are always way behind the growth and change of the Web.

In fact there is one metadata-based general purpose lookup facility: Yahoo! Yahoo doesn't use a robot. When you search through Yahoo, you're searching through human-generated subject categories and site labels. Compared to the amount of metadata that a library maintains for its books, Yahoo! is pitiful; but its popularity is clear evidence of the power of (even limited) metadata.

Pages: **1**, 2, 3 Next Page

Click Here Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate... THAWTE www.thawte.com

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

O'REILLY OPEN SOURCE CONVENTION July 23–27, 2001 San Diego conferences.oreilly.com

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

Can the samba Story be Retold?
Find it on O'Reilly Network

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

Print This Article

Using W3C XML Schema

by Eric van der Vlist
November 29, 2000

XML Schemas are an XML language for describing and constraining the content of XML documents. XML Schemas are currently in the Candidate Recommendation phase of the W3C development process.

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

Introducing Our First Schema

Let's start by having a look at this simple document which describes a book.

```
<?xml version="1.0" encoding="utf-8"?>

<book isbn="0836217462">
 <title>Being a Dog Is a Full-Time Job</title>
 <author>Charles M. Schulz</author>
 <character>
  <name>Snoopy</name>
  <friend-of>Peppermint Patty</friend-of>
  <since>1950-10-04</since>
  <qualification>
    extroverted beagle
  </qualification>
 </character>
 <character>
  <name>Peppermint Patty</name>
  <since>1966-08-22</since>
  <qualification>bold, brash and tomboyish</qualification>
 </character>
</book>
```

Get copy of library1.xml for reference.

To write a schema for this document, we could simply follow its structure and define each element as we find it. To start, we open an `xsd:schema` element.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
```

The `schema` element opens our schema. It can also hold the definition of the target namespace and several default options, of which we will see some in the following sections.

To match the start tag for the `book` element, we define an element named "book". This element has attributes and non-text children, thus we consider it a *complexType* (since the other datatype, *simpleType*, is reserved for datatypes holding only values and no element or attribute sub-nodes). The list of children of the book element is described by a *sequence* element.

```
<xsd:element name="book">
 <xsd:complexType>
```

O'REILLY OPEN SOURCE CONVENTION
July 23–27, 2001
San Diego
conferences.oreilly.com
INNOVATE
COLLABORATE
DISCOVER

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

```
  <xsd:sequence>
```

The sequence element is a *compositor* that defines an ordered sequence of sub-elements. We will see the two other compositors, *choice* and *all* in the following sections.

Now we can define the title and author elements as simple types -- they don't have attributes or non-text children and can be described directly within a degenerate *element* element. The type (xsd:string) is prefixed by the namespace prefix associated with XML Schema, indicating a predefined XML Schema datatype.

```
   <xsd:element name="title" type="xsd:string"/>
   <xsd:element name="author" type="xsd:string"/>
```

Now, we must deal with the `character` element, a complex type. Note how its cardinality is defined.

```
   <xsd:element name="character"
        minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
```

Unlike other schema definition languages, W3C XML Schema lets us define the cardinality of an element (i.e. the number of possible occurrences) with some precision. We can specify both *minOccurs* (the minimum number of occurrences) and *maxOccurs* (the maximum number of occurrences). Here, *maxOccurs* is set to "unbounded," which means that there can be as many occurrences of the character element as the author wishes. Both attributes have a default value of one.

We then specify the list of all its children in the same way.

```
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="friend-of" type="xsd:string"
            minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="since" type="xsd:date"/>
      <xsd:element name="qualification" type="xsd:string"/>
```

And we terminate its description by closing the *complexType* and *element* elements.

```
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
```

The sequence of elements for the document element (book) is now complete.

```
  </xsd:sequence>
```

We can now declare the attributes of the document elements, which must always come last. There appears to be no special reason for this, but the W3C XML Schema Working Group thought it simpler to impose a relative order to the definitions of the list of elements and attributes within a complex type, and that it was more natural to define the attributes after the elements.

```
  <xsd:attribute name="isbn" type="xsd:string"/>
```

And close all the remaining elements:

```
 </xsd:complexType>
</xsd:element>

</xsd:schema>
```

That's it! This first design, sometimes known as "Russian Doll Design," tightly follows the structure of our example document. One of the key features is to define each element and attribute within its context, and to allow multiple occurrences of a same element name to carry different definitions.

For this purpose, W3C XML Schema is a scoped language, each definition being visible only within the schema element where it is defined and all its descendants.

Here's a complete listing of this first example (download it).

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema
     xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

<xsd:element name="book">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="title" type="xsd:string"/>
```

```
    <xsd:element name="author" type="xsd:string"/>
    <xsd:element name="character"
         minOccurs="0" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="friend-of" type="xsd:string"
             minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="since" type="xsd:date"/>
       <xsd:element name="qualification" type="xsd:string"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="isbn" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

The next section explores how to subdivide schema designs to make them more readable and maintainable.

Pages: **1**, 2, 3, 4, 5, 6, 7, 8, 9

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

PORTAL WORLD.ws Opening the Windows to your Portal Solutions

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

What's On Freenet? Find it on the O'Reilly Network

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

Using W3C XML Schema
by Eric van der Vlist | Pages: 1, **2**, 3, 4, 5, 6, 7, 8, 9

Print This Article

Slicing the Schema

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

While the previous design method is very simple, it can lead to significant depth in the embedded definitions, making it hardly readable and difficult to maintain when documents are complex. It also has the drawback of being very different from a DTD structure, an obstacle for human or machine agents wishing to transform DTDs into XML Schemas, or even just use the same design guides for both technologies.

The second design is based on a flat catalog of all the elements available in the sample document and, for each of them, lists of child elements and attributes. This is achieved through using *references* to element and attribute definitions that need to be within the scope of the referencer, leading to a flat design.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

<!-- definition of simple type elements -->

<xsd:element name="title" type="xsd:string"/>
<xsd:element name="author" type="xsd:string"/>
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="friend-of" type="xsd:string"/>
<xsd:element name="since" type="xsd:date"/>
<xsd:element name="qualification" type="xsd:string"/>

<!-- definition of attributes -->

<xsd:attribute name="isbn" type="xsd:string"/>

<!-- definition of complex type elements -->

<xsd:element name="character">
 <xsd:complexType>
  <xsd:sequence>
   <!-- the simple type elements are referenced using
        the "ref" attribute                          -->
   <xsd:element ref="name"/>
   <!-- the definition of the cardinality is done
        when the elements are referenced         -->
   <xsd:element ref="friend-of"
        minOccurs="0" maxOccurs="unbounded"/>
   <xsd:element ref="since"/>
   <xsd:element ref="qualification"/>
  </xsd:sequence>
 </xsd:complexType>
```

O'REILLY OPEN SOURCE CONVENTION
July 23–27, 2001
San Diego

conferences.oreilly.com
INNOVATE COLLABORATE DISCOVER

TOOLBOX

Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

```
</xsd:element>

<xsd:element name="book">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="title"/>
   <xsd:element ref="author"/>
   <xsd:element ref="character"
        minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute ref="isbn"/>
 </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Download this schema.

Using a reference to an element or an attribute is somewhat comparable to cloning an object. The element or attribute is defined first, and it can be duplicated at another place in the document structure by the reference mechanism, in the same way an object can be cloned. The two elements (or attributes) are then two instances of the same class.

The next section shows how we can define such classes, called "types," that enable us to re-use element definitions.

Pages: 1, **2**, 3, 4, 5, 6, 7, 8, 9

Next Page

TAXI to the Future Find it on XML.com

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

Using W3C XML Schema
by Eric van der Vlist | Pages: 1, 2, **3**, 4, 5, 6, 7, 8, 9

Print This Article

Defining Named Types

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

We have seen that we can define elements and attributes as we need them (Russian doll design), or create them first and reference them (flat catalog). W3C XML Schema gives us a third mechanism, which is to define data types (either simple types that will be used for PCDATA elements or attributes, or complex types that will be used only for elements) and to use these types to define our attributes and elements.

This is achieved by giving a name to the *simpleType* and *complexType* elements, and locating them outside of the definitions of elements and attributes. We will also take the opportunity to show how we can derive a datatype from another one by defining a restriction over the values of this datatype.

For instance, to define a datatype named "nameType," which is a string with a maximum of 32 characters, we write:

```
<xsd:simpleType name="nameType">
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="32"/>
  </xsd:restriction>
 </xsd:simpleType>
```

The *simpleType* element holds the name of the new datatype. The *restriction* element expresses the fact that the datatype is derived from the "string" datatype of the W3C XML Schema namespace (attribute *base*) by applying a restriction, i.e. by limiting the number of possible values. The *maxLength* element, called a *facet*, says that this resctriction is a condition on the maximum length to be 32 characters.

Another powerful facet is the *pattern* element, which defines a regular expression that must be matched. For instance, if we do not care about "-" signs, we can define an ISBN datatype as 10 digits thus:

```
<xsd:simpleType name="isbnType">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="[0-9]{10}"/>
 </xsd:restriction>
</xsd:simpleType>
```

Facets, and the two other ways to derive a datatype (list and union), are covered further in following sections.

Complex types are defined as we've seen before, but given a name.

Defining and using named datatypes is comparable to defining a class and using it to create an object. A datatype is an abstract notion that can be used to define an attribute or an element. The datatype plays then the same role with an attribute or an element that a class would play with an object.

TOOLBOX

Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

Full listing:

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

<!-- definition of simple types -->

<xsd:simpleType name="nameType">
 <xsd:restriction base="xsd:string">
  <xsd:maxLength value="32"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="sinceType">
 <xsd:restriction base="xsd:date"/>
</xsd:simpleType>

<xsd:simpleType name="descType">
 <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="isbnType">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="[0-9]{10}"/>
 </xsd:restriction>
</xsd:simpleType>

<!-- definition of complex types -->

 <xsd:complexType name="characterType">
  <xsd:sequence>
   <xsd:element name="name" type="nameType"/>
   <xsd:element name="friend-of" type="nameType"
        minOccurs="0" maxOccurs="unbounded"/>
   <xsd:element name="since" type="sinceType"/>
   <xsd:element name="qualification" type="descType"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="bookType">
  <xsd:sequence>
   <xsd:element name="title" type="nameType"/>
   <xsd:element name="author" type="nameType"/>
   <!-- the definition of the "character" element is
        using the "characterType" complex type    -->
   <xsd:element name="character" type="characterType"
        minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="isbn" type="isbnType" use="required"/>
 </xsd:complexType>

<!-- Reference to "bookType" to define the
     "book" element -->
<xsd:element name="book" type="bookType"/>

</xsd:schema>
```

Download this schema.

The next page shows how grouping, compositors and derivation can be used to further promote re-use and structure in schemas.

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

Roll Over, Rollovers CSS Style Find it in the JavaScript / CSS DevCenter

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

Groups, Compositors and Derivation

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

Groups

W3C XML Schema also allows the definition of *groups* of elements and attributes.

```
<!-- definition of an element group -->

 <xsd:group name="mainBookElements">
  <xsd:sequence>
   <xsd:element name="title" type="nameType"/>
   <xsd:element name="author" type="nameType"/>
  </xsd:sequence>
 </xsd:group>

<!-- definition of an attribute group -->

 <xsd:attributeGroup name="bookAttributes">
  <xsd:attribute name="isbn" type="isbnType" use="required"/>
  <xsd:attribute name="available" type="xsd:string"/>
 </xsd:attributeGroup>
```

These groups can be used in the definition of complex types, as shown below.

```
<xsd:complexType name="bookType">
 <xsd:sequence>
  <xsd:group ref="mainBookElements"/>
  <xsd:element name="character" type="characterType"
       minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
 <xsd:attributeGroup ref="bookAttributes"/>
</xsd:complexType%2/>
</xsd:complexType>
```

These groups are *not* datatypes, but are *containers* holding a set of elements or attributes that can be used to describe complex types.

Compositors

So far we have seen the *xsd:sequence* compositor which defines ordered groups of elements (in fact, it defines ordered groups of *particles*, which can also be groups or other compositors). W3C XML Schema supports two additional compositors that can be mixed to allow various combinations. Each of these compositors can have *minOccurs* and *maxOccurs* attributes to define their cardinality.

The *xsd:choice* compositor describes a choice between several possible elements or groups of elements. The following group -- compositors can appear within groups, complex types or other compositors --) will accept either a single "name" element or a sequence of "firstName", an optional "middleName" and a "lastName".

```
<xsd:group name="nameTypes">
 <xsd:choice>
  <xsd:element name="name" type="xsd:string"/>
  <xsd:sequence>
   <xsd:element name="firstName" type="xsd:string"/>
   <xsd:element name="middleName" type="xsd:string" minOccurs="0"/>
   <xsd:element name="lastName" type="xsd:string"/>
  </xsd:sequence>
 </xsd:choice>
</xsd:group>
```

The *xsd:all* particle defines an unordered set of elements. The following complex type definition allows its contained elements to appear in any order:

```
<xsd:complexType name="bookType">
 <xsd:all>
  <xsd:element name="title" type="xsd:string"/>
  <xsd:element name="author" type="xsd:string"/>
  <xsd:element name="character" type="characterType"
       minOccurs="0" maxOccurs="unbounded"/>
 </xsd:all>
```

THIS SITE IS ON THE O'REILLY NETWORK

```
  <xsd:attribute name="isbn" type="isbnType" use="required"/>
 </xsd:complexType>
```

In order to avoid combinations that could become ambiguous or too complex to be solved by W3C XML Schema tools, a set of restrictions has been added to the *xsd:all* particle.

- they can appear only as a unique child at the top of a content model;
- and their children can be only *xsd:element* definitions or references, and cannot have a cardinality greater than one.

Derivation of simple types

Simple datatypes are defined by derivation from other datatypes, either predefined and identified by the W3C XML Schema namespace, or defined elsewhere in your schema.

We have already seen examples of simple types derived by restriction (using *xsd:restriction* elements). The different kind of restrictions that can be applied on a datatype are called *facets*. Beyond the *xsd:pattern* (using a regular expression syntax) and *xsd:maxLength* facets shown already, many facets allow constraints on the length of a value, an enumeration of the possible values, the minimal and maximal values, precision and scale, period and duration, etc.

Two other derivation methods are available that allow the definition of whitespace separated lists and union of datatypes. The following definition uses *xsd:union*, and extends the definition of our type for ISBN to accept the values TBD and NA.

```
<xsd:simpleType name="isbnType">
 <xsd:union>
  <xsd:simpleType>
   <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{10}"/>
   </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType>
   <xsd:restriction base="xsd:NMTOKEN">
    <xsd:enumeration value="TBD"/>
    <xsd:enumeration value="NA"/>
   </xsd:restriction>
  </xsd:simpleType>
 </xsd:union>
</xsd:simpleType>
```

The union has been applied on the two embedded simple types to allow values from both datatypes. In addition to ten digit strings, our new datatype will now accept the values from an enumeration with two possible values (TBD and NA).

The following example type (isbnTypes) uses *xsd:list* to define a whitespace-separated list of ISBN values. It also derives a type (isbnTypes8) using "xsd:restriction" that accepts between one and eight ISBN numbers, separated by whitespace.

```
<xsd:simpleType name="isbnTypes">
 <xsd:list itemType="isbnType"/>
</xsd:simpleType>

<xsd:simpleType name="isbnTypes8">
  <xsd:restriction base="isbnTypes">
   <xsd:minLength value="1"/>
   <xsd:maxLength value="8"/>
  </xsd:restriction>
</xsd:simpleType>
```

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

GPS, Palm PDA, OpenGL ... Let's Go!
Find it on the O'Reilly Network In the Wireless DevCenter

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

Print This Article

Advanced W3C XML Schema

Content Types

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

In the first part of this series we examined the default content type behavior, modeled after data-oriented documents, where complex type elements are element and attribute only, and simple type elements are character data without attributes.

The W3C XML Schema Definition Language also supports the definition of empty content elements, and simple content elements (those that contain only character data) with attributes.

Empty content elements are defined using a regular `xsd:complexType` construct and by purposefully omitting the definition of a child element. The following construct defines an empty *book* element accepting an *isbn* attribute:

```
<xsd:element name="book">
 <xsd:complexType>
  <xsd:attribute name="isbn" type="isbnType"/>
 </xsd:complexType>
</xsd:element>
```

Simple content elements, i.e. character data elements with attributes, can be derived from simple types using `xsd:simpleContent`. The book element defined above can thus be extended to accept a text value using:

```
<xsd:element name="book">
 <xsd:complexType>
  <xsd:simpleContent>
   <xsd:extension base="xsd:string">
    <xsd:attribute name="isbn" type="isbnType"/>
   </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>
</xsd:element>
```

Note the location of the attribute definition, showing that the extension is achieved through the addition of the attribute. This definition will accept the following XML element:

```
<book isbn="0836217462">
        Funny book by Charles M. Schulz.
        Its title (Being a Dog Is a Full-Time Job) says it all !
</book>
```

W3C XML Schema supports mixed content though the *mixed* attribute in the `xsd:complexType` element. Consider

```
<xsd:element name="book">
 <xsd:complexType mixed="true">
  <xsd:all>
   <xsd:element name="title" type="xsd:string"/>
   <xsd:element name="author" type="xsd:string"/>
  </xsd:all>
  <xsd:attribute name="isbn" type="xsd:string"/>
 </xsd:complexType>
</xsd:element>
```

which will validate an XML element such as

```
<book isbn="0836217462">
        Funny book by <author>Charles M. Schulz</author>.
        Its title (<title>Being a Dog Is a Full-Time Job</title>) says it all !
</book>
```

Unlike DTDs, W3C XML Schema mixed content doesn't modify the constraints on the sub-elements, which can be expressed in the same way as simple content models. While this is a significant improvement over XML 1.0 DTDs, note that the values of the character data, and its location relative to the child elements, cannot be constrained.

THIS SITE IS ON THE O'REILLY NETWORK

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

O'REILLY OPEN SOURCE CONVENTION July 23-27, 2001 San Diego conferences.oreilly.com

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

palm Shots Digital Slide Shows on Your Palm Find it on the O'Reilly Network

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List

Using W3C XML Schema
by Eric van der Vlist | Pages: 1, 2, 3, 4, 5, **6**, 7, 8, 9

Print This Article

Constraints

W3C XML Schema provides several flexible XPath-based features for describing uniqueness constraints and corresponding references constraints. The first of these, a simple uniqueness declaration, is declared with the `xsd:unique` element. The following declaration, within the context of our *book* document, indicates that the character name must be unique.

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

```
<xsd:unique name="charNameMustBeUnique">
  <xsd:selector xpath="character"/>
  <xsd:field xpath="name"/>
</xsd:unique>
```

This location of the `xsd:unique` element in the schema gives the context node in which the constraint holds. By inserting `xsd:unique` under our *book* element, we specify that the character has to be unique in the context of a book only.

The two XPaths defined in the uniqueness constraint are evaluated relative to the context node. The first of these paths is defined by the `selector` element. The purpose is to define the element which has the uniqueness constraint -- the node to which the selector points must be an element node.

The second path, specified in the `xsd:field` element. is evaluated relative to the element identified by the `xsd:selector` and can be an element or an attribute node. This is the node whose value will be checked for uniqueness. Uniqueness over a combination of several values can be specified by adding other `xsd:field` elements within `xsd:unique`.

Keys

The second constraint construct, `xsd:key`, is similar to `xsd:unique`, except that the value specified as unique can be used as a *key*. This means that it has to be non-null, and that it can be referenced. To use the character name as a key, we can replace the `xsd:unique` by `xsd:key`.

```
<xsd:key name="charNameIsKey">
  <xsd:selector xpath="character"/>
  <xsd:field xpath="name"/>
</xsd:key>
```

The third construct, `xsd:keyref`, allows us to define a reference to a key. To show its usage, we introduce the `friend-of` element, to be used against characters.

```
<character>
  <name>Snoopy</name>
  <friend-of>Peppermint Patty</friend-of>
  <since>1950-10-04</since>
  <qualification>
    extroverted beagle
```

XML
Products
Tools
Training
O'REILLY xml.com

Submissions List

TOOLBOX

Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

```
  </qualification>
 </character>
```

To indicate that `friend-of` needs to refer to a character from the same book, we write, at the same level as we defined our key constraint, the following:

```
<xsd:keyref name="friendOfIsCharRef" refer="charNameIsKey">
  <xsd:selector xpath="character"/>
  <xsd:field xpath="friend-of"/>
</xsd:keyref>
```

These capabilities are nearly independent of the other features in a schema. They are disconnected from the definition of the datatypes. The only point anchoring them to the schema is the place where they are defined, which establishes the scope of the uniqueness constraints.

Pages: 1, 2, 3, 4, 5, **6**, 7, 8, 9

Next Page

TAXI to the Future

Find it on XML.com

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

Print This Article

Building Usable -- and Reusable -- Schemas

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

Perhaps the first step in writing reusable schemas is to document them. W3C XML Schema provides an alternative to XML comments and processing instructions that might be easier to handle for supporting tools.

Human readable documentation can be defined by `xsd:documentation` elements, while information targeted at applications should be included in `xsd:appinfo` elements. Both elements must be included in an `xsd:annotation` element. They accept optional `xml:lang` and `source` attributes. The `source` attribute is a URI reference that can be used to indicate the purpose of the appinfo to the processing application.

The `xsd:annotation` elements can be added at the beginning of most schema constructs as shown in example below. The appinfo section demonstrates how custom namespaces and schemes might allow the binding of an element to a Java class from within the schema.

```
<xsd:element name="book">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Top level element.
    </xsd:documentation>
    <xsd:documentation xml:lang="fr">
      Element racine.
    </xsd:documentation>
    <xsd:appinfo source="http://example.com/foo/">
      <bind xmlns="http://example.com/bar/">
        <class name="Book"/>
      </bind>
    </xsd:appinfo>
  </xsd:annotation>
  ...
```

Composing schemas from multiple files

For those who want to define a schema using several XML documents -- either to split up a large schema or to use libraries of schema snippets -- W3C XML Schema provides two mechanisms for including external schemas.

The first, `xsd:include`, is similar to a copy and paste of the definitions of the included schema: it's an inclusion, and as such it doesn't allow any overriding of definitions of the included schema. It can be used in this way:

```
<xsd:include schemaLocation="character.xsd"/>
```

The second inclusion mechanism, `xsd:redefine`, is similar to `xsd:include`, except that it lets you redefine the declarations from the included schema.

```
<xsd:redefine schemaLocation="character12.xsd">
<xsd:simpleType name="nameType">
 <xsd:restriction base="xsd:string">
  <xsd:maxLength value="40"/>
 </xsd:restriction>
</xsd:simpleType>
</xsd:redefine>
```

Note that the declarations that are redefined *must be placed in* the `xsd:redefine` element.

We've already seen many features that can be used together with `xsd:include` and `xsd:redefine` to create libraries of schemas. We've seen how we can reference previously defined elements; how we can define datatypes by derivation and use them; and how we can define and use groups of attributes. We've also seen the parallel between elements and objects and datatypes and classes. There are other features borrowed from object oriented design that can be used to create reusable schemas.

Abstract types

The first feature derived from object oriented design is the substitution group. Unlike the features we've seen so far, a substitution group isn't defined explicitly through a W3C XML Schema element but through referencing a common element (called the *head*), using a `substitutionGroup` attribute. The head element doesn't hold any specific declaration but must be global. All the elements within a substitution group need to have a type that is either the same type as the head element, or can be derived from it. Then they can all be used in place of the head

element. In the following example the element "surname" can be used anywhere an element "name" has been defined.

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="surname" type="xsd:string" substitutionGroup="name" />
```

Now we can also define a generic "name-elt" element, head of a substitution group, that couldn't be used directly but should be used in one of its derived forms. This is done through declaring the element as abstract, analagously to abstract classes in object oriented languages. The following example defines `name-elt` as an abstract element that should be replaced by either name or surname everywhere it is referenced.

```
<xsd:element name="name-elt" type="xsd:string" abstract="true"/>
<xsd:element name="name" type="xsd:string" substitutionGroup="name-elt" />
<xsd:element name="surname" type="xsd:string" substitutionGroup="name-elt" />
```

Final types

We could, on the other hand, wish to control derivation performed on a datatype. W3C XML Schema supports this though the `final` attribute in an `xsd:complexType` or `xsd:element` element. This attribute can take the values *restriction*, *extension* and *#all* to block derivation by restriction, extension or any derivation. The following snippet would, for instance, forbid any derivation of the *characterType* complex type.

```
<xsd:complexType name="characterType" final="#all">
```

The `final` attribute can operate only on elements and complex types. W3C XML Schema provides a fine-grained mechanism that operates on each facet to control the derivation of simple types. This attribute is called `fixed`, and when its value is set to *true*, the facet cannot be further modified (but other facets can still be added or modified). The following prevents the size of our *nameType* simple type from being redefined.

```
<xsd:simpleType name="nameType">
 <xsd:restriction base="xsd:string">
  <xsd:maxLength value="32" fixed="true"/>
 </xsd:restriction>
</xsd:simpleType>
```

Pages: 1, 2, 3, 4, 5, 6, 7, 8, 9

Next Page

The End of Streaming Media
Find it on the O'Reilly Network

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker

Using W3C XML Schema
by Eric van der Vlist | Pages: 1, 2, 3, 4, 5, 6, 7, **8**, 9

Print This Article

Namespaces

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

Namespace support in W3C XML Schema is flexible yet straightforward. It not only allows the use of any prefix in instance documents (unlike DTDs), but also lets you open your schemas to accept unknown elements and attributes from known or unknown namespaces.

Each W3C XML Schema document is bound to a specific namespace through the `targetNamespace` attribute or to the absence of namespace through the lack of such an attribute. We need at least one schema document per namespace we want to define (elements and attributes without namespaces can be defined in any schema, though).

Until now we have omitted the `targetNamespace` attribute, which means that we were working without namespaces. To get into namespaces, let's imagine that our example belongs to a single namespace.

```
<book isbn="0836217462" xmlns="http://example.org/ns/books/">
```

The least intrusive way to adapt our schema is to add more attributes to our `xsd:schema` element.

```
<xsd:schema
     xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
         xmlns="http://example.org/ns/books/"
         targetNamespace="http://example.org/ns/books/"
         elementFormDefault="qualified"
         attributeFormDefault="unqualified" >
```

The namespace declarations play an important role. The first (`xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"`) says not only that we've chosen to use the prefix `xsd` to identify the elements that will be W3C XML Schema instructions, but also that we will prefix the W3C XML Schema predefined datatypes with `xsd`, as we have done in all our examples thus far. Understand that we could have chosen any prefix instead of `xsd`. We could even make *http://www.w3.org/2000/10/XMLSchema* our default namespace. In this case, we would not have prefixed the W3C XML Schema elements.

Since we are working with the *http://example.org/ns/books/* namespace, we define it as our default namespace. This means that we won't prefix the references to objects (datatypes, elements, attributes, etc.) belonging to this namespace. Again we could have chosen any prefix to identify this namespace.

The `targetNamespace` attribute lets you define, independently of the namespace declarations, which namespace is described in this schema. If you need to reference objects belonging to this namespace, which is usually the case except when using a pure Russian Doll design, you need to provide a namespace declaration in addition to the `targetNamespace`.

The final two attributes in the example, (`elementFormDefault` and `attributeFormDefault`), are a facility provided by W3C XML Schema to control, within a single schema, whether attributes and elements are considered by default to be qualified (in a namespace). This differentiation between qualified and unqualified can be indicated by specifying the default values, as above, but also when defining the element or attribute, by adding a `form` attribute of value *qualified* or *unqualified*.

It is important to note that only local elements and attributes can be specified as unqualified. All globally defined elements and attributes must always be qualified.

Importing definitions from external namespaces

W3C XML Schema, not unlike XSLT and XPath, uses namespace prefixes within the value of some attributes to identify the namespace of data types, elements, attributes, etc. For instance, we've used this feature all along in our examples to identify the W3C XML Schema predefined datatypes. This mechanism can be extended to import definitions from any other namespace and so reuse them in our schemas.

Reusing definitions from other namespaces is done through a three-step process. This process needs to be done even for the XML 1.0 namespace in order to declare attributes such as `xml:lang`. First, the namespace must be defined as usual.

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  targetNamespace="http://example.org/ns/books/"
  xmlns:xml="http://www.w3.org/XML/1998/namespace"
  elementFormDefault="qualified" >
```

Then W3C XML Schema needs to be informed of the location at which it can find the schema corresponding to the namespace. This is done using an `xsd:import` element.

```
<xsd:import namespace="http://www.w3.org/XML/1998/namespace"
  schemaLocation="myxml.xsd"/>
```

W3C XML Schema now knows that it should attempt to find any reference belonging to the XML namespace in a schema located at *myxml.xsd*. We can now use the external definition.

```
<xsd:element name="title">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute ref="xml:lang"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

You may wonder why we've chosen to reference the `xml:lang` attribute from the XML namespace rather than creating an attribute with a type `xml:lang`. We've done so because there is an important difference between referencing an attribute (or an element) and referencing a datatype when namespaces are concerned.

- Referencing an element or an attribute imports the whole thing with its name and namespace.
- Referencing a datatype imports only its definition, leaving you with the task of giving a name to the element or attribute you're defining, and places your definition in the target namespace (or no namespace if your attribute or element is unqualified).

Including unknown elements

To finish this section about namespaces, we need to see how, as promised in the introduction, we can open our schema to unknown elements, attributes and namespaces. This is done using `xsd:any` and `xsd:anyAttribute`, allowing, respectively, the inclusion of any element or attribute.

For instance, if we want to extend the definition of our description type to any XHTML tag, we could declare

```
<xsd:complexType name="descType" mixed="true">
  <xsd:sequence>
    <xsd:any namespace="http://www.w3.org/1999/xhtml"
      minOccurs="0" maxOccurs="unbounded"
      processContents="skip"/>
  </xsd:sequence>
</xsd:complexType>
```

The `xsd:anyAttribute` gives the same functionality for attribute definitions.

The type *descType* is now mixed content and accepts an unbounded number of any elements from the *http://www.w3.org/1999/xhtml namespace*. The `processContents` attribute is set to *skip*, telling a W3C XML Schema processor that no validation of these elements should be attempted. The other permissible values for this attribute are *strict*, asking to validate these elements, or *lax*, asking the processor to validate them when possible. The `namespace` attribute accepts a whitespace-separated list of URIs, as well as the special values *##any* (any namespace), *##local* (non-qualified elements), *##targetNamespace* (the target namespace) or *##other* (any namespace other than the

target).

Pages: 1, 2, 3, 4, 5, 6, 7, **8**, 9

Next Page

Resource Description Format

RDF Applications with Prolog

Find it on XML.com

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

Using W3C XML Schema
by Eric van der Vlist | Pages: 1, 2, 3, 4, 5, 6, 7, 8, **9**

Print This Article

W3C XML Schema and Instance Documents

Table of Contents

•Introducing Our First Schema
•Slicing the Schema
•Defining Named Types
•Groups, Compositors and Derivation
•Content Types
•Constraints
•Building Usable and Reusable Schemas
•Namespaces
•W3C XML Schema and Instance Documents
•W3C XML Schema Datatypes Reference
•W3C XML Schema Structures Reference

We've now covered most of the features of W3C XML Schema, but we still need to have a glance at some extensions that you can use within your instance documents. In order to differentiate these other features, a separate namespace, *http://www.w3.org/2000/10/XMLSchema-instance*, is used, usually associated with the prefix `xsi`.

The `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes allow you to tie a document to its W3C XML Schema. This link is *not* mandatory, and other indications can be given using application-dependent mechanisms (such as a parameter on a command line), but it does help W3C XML Schema aware tools to locate a schema.

Dependent on using namespaces, the link will be either

```
<book isbn="0836217462"
        xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="file:library.xsd">
```

Or, as below (noting the syntax, with a URI for the namespace and the URI of the schema separated by a whitespace in the same attribute)

```
<book isbn="0836217462" xmlns="http://example.org/ns/books/"
        xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
        xsi:schemaLocation="http://example.org/ns/books/ file:library.xsd">
```

The other use of `xsi` attributes is to provide information about how an element corresponds to a schema. These attributes are `xsi:type`, which lets you define the simple or complex type of an element, and `xsi:null`, which lets you specify a null value for an element (that has to be defined as `nullable="true"` in the schema). You don't need to declare these attributes in your schema to be able to use them in an instance document.

THIS SITE IS ON THE O'REILLY NETWORK

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

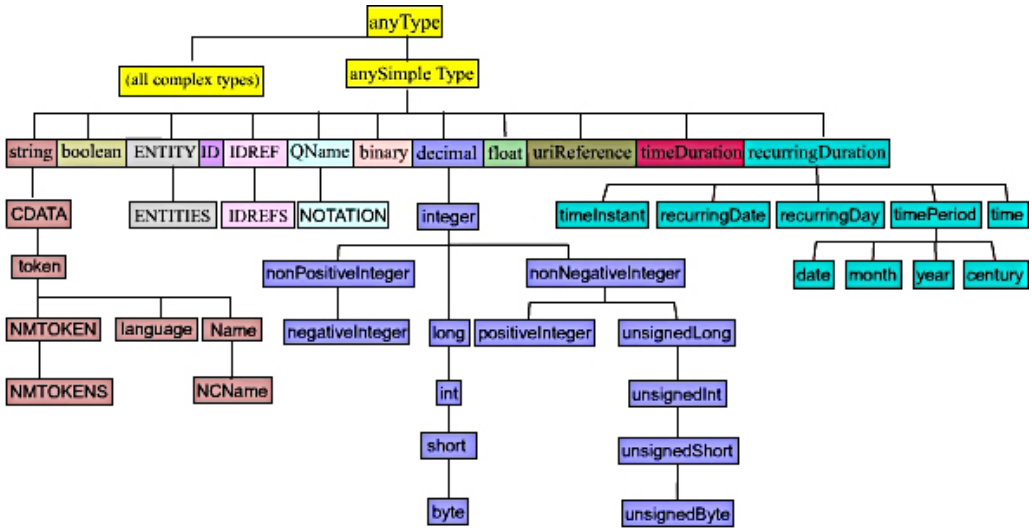
Print This Article

W3C XML Schema Datatypes Reference

by Rick Jelliffe
November 29, 2000

This quick reference helps you easily locate the definition of datatypes in the XML Schema specification. A "What You Need To Know" section gives a brief introduction to the way datatypes work.

Specification Map



What You Need To Know

- W3C XML Schema specification defines many different built-in datatypes. These datatypes can be used to constrain the values of attributes or elements which contain only simple content. These datatypes are not available for constraining data in mixed content.

Derivation and Facets

- All simple datatypes are derived from their base type by restricting the values allowed in their lexical spaces or their value spaces.
- Every datatype has a set of facets that characterize the properties of the datatype. For example, the length of a string or the encoding of a binary type (i.e., whether hex encoding or base64). By restricting some of the many facets, a new datatype can be derived.
- There are three varieties of datatypes that you can use when deriving your own datatypes: as well as atomic datatypes, where the data contains a single value, you can derive a list, where the data is treated as a whitespace-separated list of tokens, and a union type, where the lexical value of the data determines which of the base types is used.

Usage of the string datatype

The string datatype should not be used for general text. Use a complex type instead, allowing mixed content and "wildcarding" it to allow elements from other namespaces. This kind of declaration will be more future-proof. It is impossible to extend an element declared to have simple content so that it can contain sub-elements. Here is a definition that may be more suitable:

```
<complexType name="kindToStrangersText"  mixed="true" >
```

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

```
  <annotation>
    <documentation xml:lang="en" >
    This is a type definition for generic text in XML.
    For maintenance reasons, it is preferable to use
    something like this rather than the built-in datatype
    string, unless you have an absolute requirement to
    use a simple datatype.
    </documentation>
  </annotation>
  <group minOccurs="0" maxOccurs="unbounded" >
    <any namespace="##other" />
  </group>
  <attributeGroup ref="xml:specialAttrs"/>
  <anyAttribute namespace="##any" />
</complexType>
```

You will have to import the xml:lang and xml:space definitions too:

```
<import namespace="http://www.w3.org/XML/1998/namespace"
   schemaLocation="http://www.w3.org/2000/10/xml.xsd" />
```

And the schema element itself should probably have namespace declaration.

```
xmlns:xml="http://www.w3.org/XML/1998/namespace"
```

Limitations

There is no provision for

- overriding facets in the instance document,
- creating quantity/unit pairs,
- declaring n>1 dimensional arrays of tokens,
- specifying inheritance effects,
- declaring complex constraints where the value of some other information item in the instance (e.g. an attribute) has an effect on the current datatype.

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com xml from the inside out

O'REILLY OPEN SOURCE CONVENTION July 23-27, 2001 San Diego conferences.oreilly.com

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

Code + Law: An interview with Lawrence Lessig
Find it at openp2p.com

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

Print This Article

W3C XML Schema Structures Reference

by Eric van der Vlist
November 29, 2000

The quick reference below has been created using material from the W3C XML Schema Candidate Recommendation, 24 October 2000. Links to the original document are provided for each element (labeled as "ref" after each element name).

Namespaces:

- http://www.w3.org/2000/10/XMLSchema
  Namespace to be used for W3C XML Schema itself. Identified below without prefix.

- http://www.w3.org/2000/10/XMLSchema-instance
  Namespace to be used for W3C XML Schema extensions in instance documents. Identified below as "xsi".

Document instance attributes:

- xsi:noNamespaceSchemaLocation
  Location of a W3C XML Schema without target namespace.

- xsi:null
  Declaration of a null value.

- xsi:schemaLocation
  Location of a W3C XML Schema with a target namespace.

- xsi:type
  Indocument declaration of a W3C XML Schema datatype.

Elements:

- **all** (ref)
  Particle describing an unordered group of elements.

```
<all
        id = ID >
        Content: (annotation? , element*)
</all>
```

  Can be included in: complexType, group

- **annotation** (ref)
  Informative data for human or electronic agents.

```
<annotation
        {any attributes with non-schema namespace . . .}>
        Content: (appinfo |
documentation)*
</xsd:annotation>
```

  Can be included in: all, any, anyAttribute, attribute, attributeGroup, choice, complexContent, complexType, duration, element, encoding, enumeration, extension, field, group, import, include, key, keyref, length, list, maxExclusive, maxInclusive, maxLength, minExclusive, minInclusive, minLength, notation, pattern, period, precision, redefine, restriction, scale, schema, selector, sequence, simpleContent, simpleType, union, unique

- **any** (ref)
  Wildcard to replace any element.

```
<any
        id = ID
        maxOccurs = (nonNegativeInteger | unbounded)  : 1
        minOccurs = nonNegativeInteger : 1
        namespace = ((##any | ##other) | List of (uriReference | (##targetNamespace |
                ##local)) )  : ##any
        processContents = (skip | lax | strict) : strict
        {any attributes with non-schema namespace . . .}>
        Content: (annotation?)
</any>
```

  Can be included in: choice, sequence

- **anyAttribute** (ref)
  Wildcard to replace any elements.

```
<anyAttribute
        id = ID
        namespace = ((##any | ##other) | List of (uriReference | (##targetNamespace |
                ##local)) )  : ##any
        processContents = (skip | lax | strict) : strict
        {any attributes with non-schema namespace . . .}>
        Content: (annotation?)
</anyAttribute>
```

  Can be included in: attributeGroup, complexType, extension

- **appInfo** (ref)
  Information for an application.

```
<appinfo
        source = uriReference>
        Content: ({any})*
</appinfo>
```

  Can be included in:

- **attribute** (ref)
  Attribute declaration or reference.

```
<attribute
        form = (qualified | unqualified)
        id = ID
        name = NCName
        ref = QName
        type = QName
        use = (prohibited | optional | required | default | fixed) : optional
        value = string
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (simpleType?))
</attribute>
```

  Can be included in: attributeGroup, complexType, extension, schema

- **attributeGroup** (ref)
  Group of attributes.

```
<attributeGroup
        id = ID
        name = NCName
        ref = QName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , ((attribute | attributeGroup)* , anyAttribute?))
</attributeGroup>
```

  Can be included in: attributeGroup, complexType, extension, redefine, schema

- **choice** (ref)
  Particle for a group of mutually exclusive elements.

```
<choice
        id = ID
        maxOccurs = (nonNegativeInteger | unbounded)  : 1
        minOccurs = nonNegativeInteger : 1
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (element | group | choice | sequence | any)*)
</choice>
```

  Can be included in: choice, complexType, group, sequence

- **complexContent** (ref)
  Derivation of a simple type to complex content.

```
<complexContent
        id = ID
        mixed = boolean
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (restriction | extension))
</complexContent>
```

  Can be included in: complexType

- **complexType** (ref)
  Definition of or reference to a complex type.

```
<complexType
        abstract = boolean : false
        block = (#all | List of (extension | restriction))
        final = (#all | List of (extension | restriction))
        id = ID
        mixed = boolean : false
        name = NCName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (simpleContent | complexContent | ((group | all |
choice |
                sequence)? , ((attribute | attributeGroup)* , anyAttribute?))))
</complexType>
```

  Can be included in: element, redefine, schema

- **documentation** (ref)
  Human targeted documentation.

```
<documentation
        source = uriReference
        xml:lang = language>
        Content: ({any})*
</documentation>
```

  Can be included in: annotation

- **duration** (ref)
  Facet to define a duration.

```
<duration
        id = ID
        value = timeDuration
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</duration>
```

  Can be included in: restriction

- **element** (ref)
  Element declaration or reference.

```
<element
        abstract = boolean : false
        block = (#all | List of (substitution | extension | restriction))
        default = string
        final = (#all | List of (extension | restriction))
        fixed = string
        form = (qualified | unqualified)
        id = ID
        maxOccurs = (nonNegativeInteger | unbounded)  : 1
        minOccurs = nonNegativeInteger : 1
        name = NCName
        nullable = boolean : false
        ref = QName
        substitutionGroup = QName
        type = QName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , ((simpleType |
complexType)? , (key | keyref |
unique)*))
</element>
```

  Can be included in: all, choice, schema, sequence

- **encoding** (ref)
  Facet to define the encoding for binary streams.

```
<encoding
        id = ID
        value = hex | base64
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</encoding>
```

  Can be included in: restriction

- **enumeration** (ref)
  Facet to restrict a datatype to a finite set of values.

```
<enumeration
        id = ID
        value = string
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</enumeration>
```

  Can be included in: restriction

- **extension** (ref)
  Extension of a datatype.

```
<extension
        base = QName
        id = ID
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , ((attribute | attributeGroup)* , anyAttribute?))
</extension>
```

  Can be included in: complexContent, simpleContent

- **field** (ref)
  Definition of the field to be used for a uniqueness constraint.

```
<field
        id = ID
        xpath = An XPath expression
```

```
        {any attributes with non-schema namespace . . .}>
        Content: (annotation?)
</field>
```

Can be included in: key, keyref, unique

- **group** (ref)

  Definition of or reference to a group of elements.

```
<group
        id = ID
        maxOccurs = (nonNegativeInteger | unbounded)  : 1
        minOccurs = nonNegativeInteger : 1
        name = NCName
        ref = QName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (all | choice | sequence)?)
</group>
```

  Can be included in: choice, complexType, redefine, schema, sequence

- **import** (ref)

  Import of a W3C XML Schema for another namespace.

```
<import
        id = ID
        namespace = uriReference
        schemaLocation = uriReference
        {any attributes with non-schema namespace . . .}>
        Content: (annotation?)
</import>
```

  Can be included in: schema

- **include** (ref)

  Inclusion of a W3C XML Schema for the same target namespace.

```
<include
        id = ID
        schemaLocation = uriReference
        {any attributes with non-schema namespace . . .}>
        Content: (annotation?)
</include>
```

  Can be included in: schema

- **key** (ref)

  Definition of a key.

```
<key
        id = ID
        name = NCName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (selector , field+))
</key>
```

  Can be included in: element

- **keyref** (ref)

  Definition of a key reference.

```
<keyref
        id = ID
        name = NCName
        refer = QName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (selector , field+))
</keyref>
```

  Can be included in: element

- **length** (ref)

  Facet to define the length of a value.

```
<length
        id = ID
        value = nonNegativeInteger
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</length>
```

  Can be included in: restriction

- **list** (ref)

  Derivation by list.

```
<list
        id = ID
        itemType = QName
        {any attributes with non-schema namespace. . .}>
        Content: (annotation? , simpleType?)
</list>
```

  Can be included in: simpleType

- **maxExclusive** (ref)
  Facet to define a maximum (exclusive) value.

```
<maxExclusive
        id = ID
        value = string
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</maxExclusive>
```

  Can be included in: restriction

- **maxInclusive** (ref)
  Facet to define a maximum (inclusive) value.

```
<maxInclusive
        id = ID
        value = string
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</maxInclusive>
```

  Can be included in: restriction

- **maxLength** (ref)
  Facet to define a maximum length.

```
<maxLength
        id = ID
        value = string
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
<maxLength>
```

  Can be included in: restriction

- **minExclusive** (ref)
  Facet to define a minimum (exclusive) value.

```
<minExclusive
        id = ID
        value = string
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</minExclusive>
```

  Can be included in: restriction

- **minInclusive** (ref)
  Facet to define a minimum (inclusive) value.

```
<minInclusive
        id = ID
        value = string
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</minInclusive>
```

  Can be included in: restriction

- **minLength** (ref)
  Facet to define a minimum length.

```
<minLength
        id = ID
        value = string
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
<minLength>
```

  Can be included in: restriction

- **notation** (ref)
  Declaration of a notation.

```
<notation
        id = ID
        name = NCName
        public = A public identifier, per ISO 8879
        system = uriReference
        {any attributes with non-schema namespace . . .}>
        Content: (annotation?)
</notation>
```

  Can be included in: schema

- **pattern** (ref)
  Facet to define a regular expression pattern constraint.

```
<pattern
```

```
        id = ID
        value = string
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</pattern>
```

Can be included in: restriction

- **period** (ref)
  Facet to define a period.

```
<period
        id = ID
        value = timeDuration
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</period>
```

Can be included in: restriction

- **precision** (ref)
  Facet to define the precision of a numeric datatype.

```
<precision
        id = ID
        value = nonNegativeInteger
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</precision>
```

Can be included in: restriction

- **redefine** (ref)
  Import of a W3C XML Schema for the same namespace with possible overide.

```
<redefine
        schemaLocation = uriReference
        {any attributes with non-schema namespace . . .}>
        Content: (annotation | (attributeGroup | complexType | group | simpleType))*
</redefine>
```

Can be included in: schema

- **restriction** (ref)
  Derivation of a simple datatype by restriction.

```
<restriction
        id = ID
        base = QName
        {any attributes with non-schema namespace. . .}>
        Content: (annotation? , (simpleType? ,
(minExclusive | minInclusive |
maxExclusive | maxInclusive |
        precision | scale | length |
minLength | maxLength |
encoding | period | duration |
enumeration | pattern)*))
</restriction>
```

Can be included in: complexContent, simpleContent, simpleType

- **scale** (ref)
  Facet to define the scale of a numeric datatype.

```
<scale
        id = ID
        value = nonNegativeInteger
        fixed = boolean : false
        {any attributes with non-schema namespace. . .}>
        Content: (annotation?)
</scale>
```

Can be included in: restriction

- **schema** (ref)
  Document element of a W3C XML Schema.

```
<schema
        attributeFormDefault = (qualified | unqualified) : unqualified
        blockDefault = (#all | List of (substitution | extension | restriction))
        elementFormDefault = (qualified | unqualified) : unqualified
        finalDefault = (#all | List of (extension | restriction))
        id = ID
        targetNamespace = uriReference
        version = string
        {any attributes with non-schema namespace . . .}>
        Content: ((include | import |
redefine | annotation)* ,
((attribute | attributeGroup    |
complexType | element |
```

```
group | notation | simpleType) ,
annotation*)*)
</schema>
```

Can be included in:

- **selector** (ref)
  Definition of the the path selecting an element for a uniqueness constraint.

```
<selector
        id = ID
        xpath = An XPath expression
        {any attributes with non-schema namespace . . .}>
        Content: (annotation?)
</selector>
```

  Can be included in: key, keyref, unique

- **sequence** (ref)
  Particle to define an ordered group of elements.

```
<sequence
        id = ID
        maxOccurs = (nonNegativeInteger | unbounded)  : 1
        minOccurs = nonNegativeInteger : 1
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (element | group | choice | sequence | any)*)
</sequence>
```

  Can be included in: choice, complexType, group, sequence

- **simpleContent** (ref)
  Simple content declaration for an element.

```
<simpleContent
        id = ID
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (restriction | extension))
</simpleContent>
```

  Can be included in: complexType

- **simpleType** (ref)
  Simple type declaration.

```
<simpleType
        id = ID
        name = NCName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , ((list | restriction | union)))
</simpleType>
```

  Can be included in: attribute, element, list, redefine, restriction, schema, union

- **union** (ref)
  Derivation of datatypes by union.

```
<union
        id = ID
        memberTypes = List of QName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (simpleType*))
</union>
```

  Can be included in: simpleType

- **unique** (ref)
  Definition of a uniqueness constraint.

```
<unique
        id = ID
        name = NCName
        {any attributes with non-schema namespace . . .}>
        Content: (annotation? , (selector , field+))
</unique>
```

  Can be included in: element

Portions of this document are Copyright © 1999, 2000 W3C® (MIT, INRIA, Keio)

Click Here
Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate...
THAWTE
www.thawte.com

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Pre-conference workshops: 25th July
Conference: 26th&27th July
Holiday Inn London, Victoria
Register Now

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

FIND
Search
Article Archive
FAQs

The Annotated XML Specification

Print This Article

by C.M. Sperberg-McQueen, Jean Paoli, Tim Bray
April 15, 1998

Inside the XML 1.0 Specification

If you want to understand XML, you have to read the specification. However, to really get inside the specification and understand why it says what it does, you need an expert guide. Tim Bray, co-editor of the XML 1.0 specification, shares his knowledge and insights about XML, SGML and the working group behind the specification in this annotated version of the document.

Tim created the Annotated XML Specification in XML, and wrote an excellent explanation of how he did this.

Clicking on the link below will open the Annotated XML Specification in a frameset window, along with a floating navigation window if your browser supports JavaScript. Alternatively, you can use a three-paned frames version of the document. Use the links in the navigation window to get around the main document, as well as to return to this page, or to XML.com.

The Annotated XML 1.0 Specification

Non-JavaScript Version

Add the latest news from XML.com to your web site
Click here!

(still requires frames)

COLUMNS

XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES

XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX

Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

Click Here
Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate...
THAWTE www.thawte.com

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Pre-conference workshops: 25th July
Conference: 26th&27th July
Holiday Inn London, Victoria
Register Now

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter

search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

Java Performance-
Efficiently Formatting Doubles
Find it at ONJava.com

FIND
Search
Article Archive
FAQs

Building the Annotated XML Specification

Print This Article

by Tim Bray
September 12, 1998

The design of XML 1.0 stretched over 20 months ending in February 1998, with input from a couple of hundred of the world's best experts in the area of markup, publishing, and Web design. The result of that work, the XML 1.0 Specification, is a highly condensed document that contains little or no information about how it came to read the way it does.

Even before the release of XML 1.0, it became obvious that some parts of the spec were self-explanatory, while others were causing headaches for its users.

The Annotated XML Specification addresses both of these problems. It supplements the basic specification, first with historical background and explanation of how things came to be the way they are, and second with detailed explanations of the portions of the spec that have proved difficult. Commercially, it has been a success; in its first month on the Web, it had over 100,000 page views from over 26,000 unique Internet addresses. It remains, by a substantial margin, the most popular item available at the XML.com site.

This article explains how I created the Annotated XML Specification. If you haven't looked at it, you might want to give it a glance before reading about it, or even better, open it in another browser window while you read

Add the latest news from XML.com to your web site
Click here!

about it here.

Pages: **1**, 2, 3, 4

COLUMNS

XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES

XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX

Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

Click Here
Click Here for Your FREE Guide to Securing Your Apache Server with a Thawte SSL Certificate...
THAWTE www.thawte.com

Copyright © 2001 O'Reilly & Associates, Inc.

REC-xml-19980210

Extensible Markup Language (XML) 1.0

W3C Recommendation 10-February-1998

This version:

http://www.w3.org/TR/1998/REC-xml-19980210

http://www.w3.org/TR/1998/REC-xml-19980210.xml

http://www.w3.org/TR/1998/REC-xml-19980210.html

http://www.w3.org/TR/1998/REC-xml-19980210.pdf

http://www.w3.org/TR/1998/REC-xml-19980210.ps (U)

Latest version:

http://www.w3.org/TR/REC-xml

Previous version:

http://www.w3.org/TR/PR-xml-971208 (H)

Editors:

Tim Bray (Textuality and Netscape) <tbray@textuality.com> (H)

Jean Paoli (Microsoft) <jeanpa@microsoft.com> (H)

C. M. Sperberg-McQueen (University of Illinois at Chicago) <cmsmcq@uic.edu> (H)

Abstract

The Extensible Markup Language (H) (XML) is a subset of SGML (H) that is completely described in this document. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

Status of this document

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. (H) It is a stable document and may be used as reference material or cited as a normative reference from another document. (U) W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document specifies a syntax created by subsetting an existing, widely used international text processing standard (Standard Generalized Markup Language, ISO 8879:1986(E) as amended and corrected) for use on the World Wide Web. It is a product of the W3C XML Activity, details of which can be found at http://www.w3.org/XML. A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

This specification uses the term URI, (T) which is defined by [Berners-Lee et al.], a work in progress expected to update [IETF RFC1738] and [IETF RFC1808].

The list of known errors in this specification is available at http://www.w3.org/XML/xml-19980210-errata.

Please report errors in this document to xml-editor@w3.org. (U)

Extensible Markup Language (XML) 1.0

Table of Contents

1. Introduction
   1.1 Origin and Goals
   1.2 Terminology
2. Documents
   2.1 Well-Formed XML Documents
   2.2 Characters
   2.3 Common Syntactic Constructs
   2.4 Character Data and Markup

Introduction to the Annotated XML Specification

by Tim Bray

The other window contains the XML specification; this window the commentary on it. The content and appearance of the XML spec are exactly as in the official version; it has not been edited in any way to generate this presentation. The commentary is contained in external XML files, with XML hyperlinks into the (entirely unaltered) XML version of the spec. The footnoted HTML version that you see on the screen is program-generated. The annotations are flagged as follows:

(H) Historical or cultural commentary; some entertainment value.

(T) Technical explanations, including amplifications, corrections, and answers to Frequently Asked Questions.

(U) Advice on how to use this specification.

(E) Examples to illustrate what the spec is saying.

(M) Annotations that it's hard to find a category for.

Copyright © 1998, Tim Bray. All rights reserved.

2.5 Comments
2.6 Processing Instructions
2.7 CDATA Sections
2.8 Prolog and Document Type Declaration
2.9 Standalone Document Declaration
2.10 White Space Handling
2.11 End-of-Line Handling
2.12 Language Identification
3. Logical Structures
3.1 Start-Tags, End-Tags, and Empty-Element Tags
3.2 Element Type Declarations
3.2.1 Element Content
3.2.2 Mixed Content
3.3 Attribute-List Declarations
3.3.1 Attribute Types
3.3.2 Attribute Defaults
3.3.3 Attribute-Value Normalization
3.4 Conditional Sections
4. Physical Structures
4.1 Character and Entity References
4.2 Entity Declarations
4.2.1 Internal Entities
4.2.2 External Entities
4.3 Parsed Entities
4.3.1 The Text Declaration
4.3.2 Well-Formed Parsed Entities
4.3.3 Character Encoding in Entities
4.4 XML Processor Treatment of Entities and References
4.4.1 Not Recognized
4.4.2 Included
4.4.3 Included If Validating
4.4.4 Forbidden
4.4.5 Included in Literal
4.4.6 Notify
4.4.7 Bypassed
4.4.8 Included as PE
4.5 Construction of Internal Entity Replacement Text
4.6 Predefined Entities
4.7 Notation Declarations
4.8 Document Entity
5. Conformance
5.1 Validating and Non-Validating Processors
5.2 Using XML Processors
6. Notation

Appendices

A. References
A.1 Normative References
A.2 Other References
B. Character Classes
C. XML and SGML (Non-Normative)
D. Expansion of Entity and Character References (Non-Normative)
E. Deterministic Content Models (Non-Normative)
F. Autodetection of Character Encodings (Non-Normative)
G. W3C XML Working Group (Non-Normative)

---

1. Introduction

Extensible Markup Language, abbreviated XML, describes a class of data objects(M) called XML documents(U) and partially describes the behavior of computer programs which process them.(U) XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]. By construction, XML documents are conforming SGML documents. (U)

XML documents are made up of storage units called entities, which contain either parsed or unparsed data.(U)

Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

[Definition:] A software module called an **XML processor** is used to read XML documents and provide access to their content and structure. [Definition:] It is assumed that an XML processor is doing its work on behalf of another module, called the **application**. This specification describes the required behavior of an XML processor in terms of how it must read XML data and the information it must provide to the application.

1.1 Origin and Goals

XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996. It was chaired by Jon Bosak of Sun Microsystems with the active participation of an XML Special Interest Group (previously known as the SGML Working Group) also organized by the W3C. The membership of the XML Working Group is given in an appendix. Dan Connolly served as the WG's contact with the W3C.

The design goals for XML are:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

This specification, together with associated standards (Unicode and ISO/IEC 10646 for characters, Internet RFC 1766 for language identification tags, ISO 639 for language name codes, and ISO 3166 for country name codes), provides all the information necessary to understand XML Version 1.0 and construct computer programs to process it.

This version of the XML specification may be distributed freely, as long as all text and legal notices remain intact.

1.2 Terminology

The terminology used to describe XML documents is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of an XML processor:

may

[Definition:] Conforming documents and XML processors are permitted to but need not behave as described.

must

Conforming documents and XML processors are required to behave as described; otherwise they are in error.

error

[Definition:] A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it.

fatal error

[Definition:] An error which a conforming XML processor must detect and report to the application. After encountering a fatal error, the processor may continue processing the data to search for further errors and may report such errors to the application. In order to support correction of errors, the processor may make unprocessed data from the document (with intermingled character data and markup) available to the application. Once a fatal error is detected, however, the processor must not continue normal processing (i.e., it must not continue to pass character data and information about the document's logical structure to the application in the normal way).

at user option

Conforming software may or must (depending on the modal verb in the sentence) behave as described; if it does, it must provide users a means to enable or disable the behavior described.

validity constraint

A rule which applies to all valid XML documents. Violations of validity constraints are errors; they must, at user option, be reported by validating XML processors.

well-formedness constraint

A rule which applies to all well-formed XML documents. Violations of well-formedness constraints are fatal errors.

match

[Definition:] (Of strings or names:) Two strings or names being compared must be identical. Characters with multiple possible representations in ISO/IEC 10646 (e.g. characters with both precomposed and base+diacritic forms) match only if they have the same representation in both strings. At user option, processors may normalize such characters to some canonical form. No case folding is performed. (Of strings and rules in the grammar:) A string matches a grammatical production if it belongs to the language generated by that production. (Of content and content models:) An element matches its declaration when it conforms in the fashion described in the constraint "Element Valid".

for compatibility

[Definition:] A feature of XML included solely to ensure that XML remains compatible with SGML.

for interoperability

[Definition:] A non-binding recommendation included to increase the chances that XML documents can be processed by the existing installed base of SGML processors which predate the WebSGML Adaptations Annex to ISO 8879.

2. Documents

[Definition:] A data object is an **XML document** if it is well-formed, as defined in this specification. A well-formed XML document may in addition be valid if it meets certain further constraints.

Each XML document has both a logical and a physical structure. Physically, the document is composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. A document begins in a "root" or document entity. Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup. The logical and physical structures must nest properly, as described in "4.3.2 Well-Formed Parsed Entities".

2.1 Well-Formed XML Documents

[Definition:] A textual object is a well-formed XML document if:

1. Taken as a whole, it matches the production labeled `document`.
2. It meets all the well-formedness constraints given in this specification.
3. Each of the parsed entities which is referenced directly or indirectly within the document is well-formed.

Document
[1] `document ::= prolog element Misc*`

Matching the `document` production implies that:

1. It contains one or more elements.
2. [Definition:] There is exactly one element, called the **root**, or document element, no part of which appears in the content of any other element. For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. More simply stated, the elements, delimited by start- and end-tags, nest properly within each other.

[Definition:] As a consequence of this, for each non-root element C in the document, there is one other element P in the document such that C is in the content of P, but is not in the content of any other element that is in the content of P. P is referred to as the **parent** of C, and C as a **child** of P.

2.2 Characters

[Definition:] A parsed entity contains **text**, a sequence of characters, which may represent markup or character data. [Definition:] A **character** is an atomic unit of text as specified by ISO/IEC 10646 [ISO/IEC 10646]. Legal characters are tab, carriage return, line feed, and the legal graphic characters of Unicode and ISO/IEC 10646. The use of "compatibility characters", as defined in section 6.8 of [Unicode], is discouraged.

Character Range

```
[2] Char ::= #x9 | #xA | #xD       /* any
           | [#x20-#xD7FF]           Unicode
           | [#xE000-#xFFFD]         character,
           | [#x10000-#x10FFFF]      excluding
                                     the
                                     surrogate
                                     blocks,
                                     FFFE, and
                                     FFFF. */
```

The mechanism for encoding character code points into bit patterns may vary from entity to entity. All XML processors must accept the UTF-8 and UTF-16 encodings of 10646; the mechanisms for signaling which of the two is in use, or for bringing other encodings into play, are discussed later, in "4.3.3 Character Encoding in Entities".

2.3 Common Syntactic Constructs

This section defines some symbols used widely in the grammar.

S (white space) consists of one or more space (#x20) characters, carriage returns, line feeds, or tabs.

White Space

```
[3] S ::= (#x20 | #x9 | #xD | #xA)+
```

Characters are classified for convenience as letters, digits, or other characters. Letters consist of an alphabetic or syllabic base character possibly followed by one or more combining characters, or of an ideographic character. Full definitions of the specific characters in each class are given in "B. Character Classes".

[Definition:] A **Name** is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters. Names beginning with the string "`xml`", or any string which would match `(('X'|'x') ('M'|'m') ('L'|'l'))`, are reserved for standardization in this or future versions of this specification.

Note: The colon character within XML names is reserved for experimentation with name spaces. Its meaning is expected to be standardized at some future point, at which point those documents using the colon for experimental purposes may need to be updated. (There is no guarantee that any name-space mechanism adopted for XML will in fact use the colon as a name-space delimiter.) In practice, this means that authors should not use the colon in XML names except as part of name-space experiments, but that XML processors should accept the colon as a name character.

An `Nmtoken` (name token) is any mixture of name characters.

Names and Tokens

```
[4] NameChar ::= Letter | Digit | '.' | '-'
               | '_' | ':' | CombiningChar
               | Extender
[5]     Name ::= (Letter | '_' | ':')
               (NameChar)*
[6]    Names ::= Name (S Name)*
[7]  Nmtoken ::= (NameChar)+
[8] Nmtokens ::= Nmtoken (S Nmtoken)*
```

Literal data is any quoted string not containing the quotation mark used as a delimiter for that string. Literals are used for specifying the content of internal entities (`EntityValue`), the values of attributes (`AttValue`), and external identifiers (`SystemLiteral`). Note that a `SystemLiteral` can be parsed without scanning for markup.

Literals

```
 [9]   EntityValue ::= '"' ([^%&"]
                     | PEReference
                     | Reference)* '"'
                     |  "'" ([^%&']
                     | PEReference
                     | Reference)* "'"
[10]      AttValue ::= '"' ([^<&"]
                     | Reference)* '"'
                     |  "'" ([^<&']
                     | Reference)* "'"
[11] SystemLiteral ::= ('"' [^"]* '"') | ("'"
                     [^']* "'")
[12]  PubidLiteral ::= '"' PubidChar* '"' | "'"
                     (PubidChar - "'")* "'"
[13]     PubidChar ::= #x20 | #xD | #xA
                     | [a-zA-Z0-9]
                     | [-'()+,./:=?;!*#@$_%]
```

2.4 Character Data and Markup

Text consists of intermingled character data and markup. [Definition:] **Markup** takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, and processing instructions.

[Definition:] All text that is not markup constitutes the **character data** of the document.

The ampersand character (&) and the left angle bracket (<) may appear in their literal form *only* when used as markup delimiters, or within a comment, a processing instruction, or a CDATA section. They are also legal within the literal entity value of an internal entity declaration; see "4.3.2 Well-Formed Parsed Entities". If they are needed elsewhere, they must be escaped using either numeric character references or the strings "`&`" and "`<`" respectively. The right angle bracket (>) may be represented using the string "`>`", and must, for compatibility, be escaped using "`>`" or a character reference when it appears in the string "`]]>`" in content, when that string is not marking the end of a CDATA section.

In the content of elements, character data is any string of characters which does not contain the start-delimiter of any markup. In a CDATA section, character data is any string of characters not including the CDATA-section-close delimiter, "`]]>`".

To allow attribute values to contain both single and double quotes, the apostrophe or single-quote character (') may be represented as "`'`", and the double-quote character (") as "`"`".

Character Data

```
[14] CharData ::= [^<&]* - ([^<&]* ']]>'
                  [^<&]*)
```

2.5 Comments

[Definition:] **Comments** may appear anywhere in a document outside other markup; in addition, they may appear within the document type declaration at places allowed by the grammar. They are not part of the document's character data; an XML processor may, but need not, make it possible for an application to retrieve the text of comments. For compatibility, the string "`--`" (double-hyphen) must not occur within comments.

Comments

```
[15] Comment ::= '<!--' ((Char - '-') | ('-'
                 (Char - '-')))* '-->'
```

An example of a comment:

```
<!-- declarations for <head> & <body> -->
```

2.6 Processing Instructions

[Definition:] **Processing instructions** (PIs) allow documents to contain instructions for applications.

Processing Instructions

```
[16]       PI ::= '<?' PITarget (S (Char* -
                  (Char* '?>' Char*)))? '?>'
[17] PITarget ::= Name - (('X' | 'x') ('M'
                  | 'm') ('L' | 'l'))Ⓤ
```

PIs are not part of the document's character data, but must be passed through to the application. The PI begins with a target (`PITarget`) used to identify the application to which the instruction is directed. The target names "`XML`", "`xml`", and so on are reserved for standardization in this or future versions of this specification. The XML Notation mechanism may be used for formal declaration of PI targets.

2.7 CDATA Sections

[Definition:] **CDATA sections** may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup.Ⓣ CDATA sections begin with the string "`<![CDATA[`" and end with the string "`]]>`":Ⓣ

CDATA Sections

```
[18]  CDSect ::= CDStart CData CDEnd
[19] CDStart ::= '<![CDATA['
[20]   CData ::= (Char* - (Char* ']]>' Char*))
[21]   CDEnd ::= ']]>'
```

Within a CDATA section, only the `CDEnd` string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form; they need not (and cannot) be escaped using "`<`" and "`&`". CDATA sections cannot nest.

An example of a CDATA section, in which "`<greeting>`" and "`</greeting>`" are recognized as character data, not markup:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

2.8 Prolog and Document Type Declaration

[Definition:] XML documents may, and should,Ⓗ begin with an **XML declaration** which specifies the version of XML being used.Ⓣ For example, the following is a complete XML document, well-formed but not valid:Ⓣ

```
<?xml version="1.0"?>
<greeting>Hello, world!</greeting>
```

and so is this:

```
<greeting>Hello, world!</greeting>
```

The version number "`1.0`" should be used to indicate conformance to this version of this specification; it is an error for a document to use the value "`1.0`" if it does not conform to this version of this specification. It is the intent of the XML working group to give later versions of this specification numbers other than "`1.0`", but this intent does not indicate a commitmentⓊ to produce any future versions of XML, nor if any are produced, to use any particular numbering scheme. Since future versions are not ruled out, this construct is provided as a means to allow the possibility of automatic version recognition, should it become necessary. Processors may signal an error if they receive documents labeled with versions they do not support.

The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute-value pairs with its logical structures. XML provides a mechanism, the document type declarationⓊ, to define constraints on the logical structure and to support the use of predefined storage units. [Definition:] An XML document is **valid** if it has an associated document type declarationⓉ and if the document complies with the constraints expressed in it.Ⓣ

The document type declaration must appear before the first element in the document.

Prolog

```
[22]      prolog ::= XMLDecl?(T) Misc*
                     (doctypedecl Misc*)?
[23]    XMLDecl ::= '<?xml' VersionInfo
                     EncodingDecl? SDDecl? S?
                     '?>'(T)
[24] VersionInfo ::= S 'version' Eq ('
                     VersionNum ' | "
                     VersionNum ")(U)
[25]          Eq ::= S? '=' S?
[26]  VersionNum ::= ([a-zA-Z0-9_.:] | '-')+
[27]        Misc ::= Comment | PI |  S
```

[Definition:] The XML **document type declaration** contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition,(U) or **DTD**. The document type declaration can point to an external subset (a special kind of external entity)(T) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together.

[Definition:] A **markup declaration** is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration. These declarations may be contained in whole or in part within parameter entities, as described in the well-formedness and validity constraints below. For fuller information, see "4. Physical Structures".

Document Type Definition

```
[28] doctypedecl ::= '<!DOCTYPE'(U) [ VC: Root
                     S Name (S         Element Type ]
                     ExternalID)?
                     S? ('['
                     (markupdecl
                     | PEReference
                     | S)* ']' S?)?
                     '>'
[29]  markupdecl ::= elementdecl    [ VC: Proper
                     | AttlistDecl    Declaration/PE
                     | EntityDecl     Nesting ]
                     | NotationDecl
                     | PI | Comment
                                    [ WFC: PEs in
                                      Internal
                                      Subset ]
```

The markup declarations may be made up in whole or in part of the replacement text of parameter entities. The productions later in this specification for individual nonterminals (`elementdecl`, `AttlistDecl`, and so on) describe the declarations *after* all the parameter entities have been included.(U)

Validity Constraint: Root Element Type
The `Name` in the document type declaration must match the element type of the root element.

Validity Constraint: Proper Declaration/PE Nesting
Parameter-entity replacement text must be properly nested with markup declarations. That is to say, if either the first character or the last character of a markup declaration (`markupdecl` above) is contained in the replacement text for a parameter-entity reference, both must be contained in the same replacement text. (E)

Well-Formedness Constraint: PEs in Internal Subset
In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.) (T)

Like the internal subset, the external subset and any external parameter entities referred to in the DTD must consist of a series of complete markup declarations of the types allowed by the non-terminal symbol `markupdecl`, interspersed with white space or parameter-entity references. However, portions of the contents of the external subset or of external parameter entities may conditionally be ignored by using the conditional section construct; this is not allowed(T) in the internal subset.

External Subset

```
[30]     extSubset ::= TextDecl? extSubsetDecl
                       Ⓣ
[31] extSubsetDecl ::= ( markupdecl
                       | conditionalSect
                       | PEReference | S )*
```

The external subset and external parameter entities also differ from the internal subset in that in them, parameter-entity references are permitted *within* markup declarations, not only *between* markup declarations.

An example of an XML document with a document type declaration:

```
<?xml version="1.0"?>
<!DOCTYPE greeting SYSTEM "hello.dtd">
<greeting>Hello, world!</greeting>
```

The system identifier "`hello.dtd`" gives the URI of a DTD for the document. Ⓣ

The declarations can also be given locally, Ⓣ as in this example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

If both the external and internal subsets are used, the internal subset is considered to occur before the external subset. Ⓣ This has the effect that entity and attribute-list declarations in the internal subset take precedence over those in the external subset.

2.9 Standalone Document Declaration

Markup declarations can affect the content of the document, as passed from an XML processor to an application; examples are attribute defaults and entity declarations. The standalone document declaration, which may appear as a component of the XML declaration, signals whether or not there are such declarations which appear external to the document entity. Ⓣ

Standalone Document Declaration

```
[32] SDDecl ::= S 'standalone'  [VC: Standalone
                Eq (("'" ('yes'  Document
                | 'no') "'")     Declaration ]
                | ('"' ('yes'
                | 'no') '"'))
```

In a standalone document declaration, the value "`yes`" indicates that there are no markup declarations external to the document entity (either in the DTD external subset, or in an external parameter entity referenced from the internal subset) which affect the information passed from the XML processor to the application. The value "`no`" indicates that there are or may be such external markup declarations. Note that the standalone document declaration only denotes the presence of external *declarations*; the presence, in a document, of references to external *entities*, when those entities are internally declared, does not change its standalone status.

If there are no external markup declarations, the standalone document declaration has no meaning. If there are external markup declarations but there is no standalone document declaration, the value "`no`" is assumed.

Any XML document for which `standalone="no"` holds can be converted algorithmically to a standalone document, which may be desirable for some network delivery applications.

Validity Constraint: Standalone Document Declaration
The standalone document declaration must have the value "`no`" if Ⓣ any external markup declarations contain declarations of:

- attributes with default values, if elements to which these attributes apply appear in the document without specifications of values for these attributes, or
- entities (other than `amp`, `lt`, `gt`, `apos`, `quot`), if references to those entities appear in the document, or
- attributes with values subject to normalization, where the attribute appears in the document with a value which will change as a result of normalization, or
- element types with element content, if white space occurs directly within any instance of those types.

An example XML declaration with a standalone document declaration:

```
<?xml version="1.0" standalone='yes'?>
```

2.10 White Space Handling

In editing XML documents, it is often convenient to use "white space" (spaces, tabs, and blank lines, denoted by the nonterminal S in this specification) to set apart the markup for greater readability. Such white space is typically not intended for inclusion in the delivered version of the document. On the other hand, "significant" white space that should be preserved in the delivered version is common, for example in poetry and source code.

An XML processor must always pass all characters in a document that are not markup through to the application. A validating XML processor must also inform the application which of these characters constitute white space appearing in element content.

A special attribute named `xml:space` may be attached to an element to signal an intention that in that element, white space should be preserved by applications. In valid documents, this attribute, like any other, must be declared if it is used. When declared, it must be given as an enumerated type whose only possible values are "`default`" and "`preserve`". For example:

```
    <!ATTLIST poem   xml:space (default|preserve) 'preserve'>
```

The value "`default`" signals that applications' default white-space processing modes are acceptable for this element; the value "`preserve`" indicates the intent that applications preserve all the white space. This declared intent is considered to apply to all elements within the content of the element where it is specified, unless overriden with another instance of the `xml:space` attribute.

The root element of any document is considered to have signaled no intentions as regards application space handling, unless it provides a value for this attribute or the attribute is declared with a default value.

2.11 End-of-Line Handling

XML parsed entities are often stored in computer files which, for editing convenience, are organized into lines. These lines are typically separated by some combination of the characters carriage-return (#xD) and line-feed (#xA).

To simplify the tasks of applications, wherever an external parsed entity or the literal entity value of an internal parsed entity contains either the literal two-character sequence "#xD#xA" or a standalone literal #xD, an XML processor must pass to the application the single character #xA. (This behavior can conveniently be produced by normalizing all line breaks to #xA on input, before parsing.)

2.12 Language Identification

In document processing, it is often useful to identify the natural or formal language in which the content is written. A special attribute named `xml:lang` may be inserted in documents to specify the language used in the contents and attribute values of any element in an XML document. In valid documents, this attribute, like any other, must be declared if it is used. The values of the attribute are language identifiers as defined by [IETF RFC 1766], "Tags for the Identification of Languages":

Language Identification

```
[33] LanguageID ::= Langcode ('-' Subcode)*
[34]   Langcode ::= ISO639Code |  IanaCode |
                    UserCode
[35] ISO639Code ::= ([a-z] | [A-Z]) ([a-z]
                    | [A-Z])
[36]   IanaCode ::= ('i' | 'I') '-' ([a-z]
                    | [A-Z])+
[37]   UserCode ::= ('x' | 'X') '-' ([a-z]
                    | [A-Z])+
[38]    Subcode ::= ([a-z] | [A-Z])+
```

The `Langcode` may be any of the following:

- a two-letter language code as defined by [ISO 639], "Codes for the representation of names of languages"
- a language identifier registered with the Internet Assigned Numbers Authority [IANA]; these begin with the prefix "`i-`" (or "`I-`")
- a language identifier assigned by the user, or agreed on between parties in private use; these must begin with the prefix "`x-`" or "`X-`" in order to ensure that they do not conflict with names later standardized or registered with IANA

There may be any number of `Subcode` segments; if the first subcode segment exists and the Subcode consists of two letters, then it must be a country code from [ISO 3166], "Codes for the representation of names of countries." If the first subcode consists of more than two letters, it must be a subcode for the language in question registered with IANA, unless the `Langcode` begins with the prefix "`x-`" or "`X-`".

It is customary to give the language code in lower case, and the country code (if any) in upper case. Note that these values, unlike other names in XML documents, are case insensitive.

For example:

```
<p xml:lang="en">The quick brown fox jumps over the lazy dog.</p>
<p xml:lang="en-GB">What colour is it?</p>
<p xml:lang="en-US">What color is it?</p>
<sp who="Faust" desc='leise' xml:lang="de">
  <l>Habe nun, ach! Philosophie,</l>
  <l>Juristerei, und Medizin</l>
  <l>und leider auch Theologie</l>
  <l>durchaus studiert mit heißem Bemüh'n.</l>
  </sp>
```

The intent declared with `xml:lang` is considered to apply to all attributes and content of the element where it is specified, unless overridden with an instance of `xml:lang` on another element within that content.

A simple declaration for `xml:lang` might take the form

```
xml:lang  NMTOKEN  #IMPLIED
```

but specific default values may also be given, if appropriate. In a collection of French poems for English students, with glosses and notes in English, the xml:lang attribute might be declared this way:

```
    <!ATTLIST poem   xml:lang NMTOKEN 'fr'>
    <!ATTLIST gloss  xml:lang NMTOKEN 'en'>
    <!ATTLIST note   xml:lang NMTOKEN 'en'>
```

3. Logical Structures

[Definition:] Each XML document contains one or more **elements**,(E) the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type,(U) identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications. Each attribute specification has a name and a value.

Element	
[39] element ::= EmptyElemTag | STag content ETag	[WFC: Element Type Match] [VC: Element Valid]

This specification does not constrain the semantics, use, or (beyond syntax) names of the element types and attributes,(H) except that names beginning with a match to `(('X'|'x')('M'|'m')('L'|'l'))` are reserved for standardization in this or future versions of this specification.

Well-Formedness Constraint: Element Type Match
The `Name` in an element's end-tag must match the element type in the start-tag.

Validity Constraint: Element Valid(U)
An element is valid if there is a declaration matching `elementdecl` where the `Name` matches the element type,(T) and one of the following holds:

1. The declaration matches `EMPTY` and the element has no content.(T)
2. The declaration matches `children` and the sequence of child elements belongs to the language generated by the regular expression in the content model, with optional white space (characters matching the nonterminal S) between each pair of child elements.(T)
3. The declaration matches `Mixed` and the content consists of character data and child elements whose types match names in the content model.(T)
4. The declaration matches `ANY`, and the types of any child elements have been declared.(T)

3.1 Start-Tags, End-Tags, and Empty-Element Tags

[Definition:] The beginning of every non-empty XML element is marked by a **start-tag**.

Start-tag

```
[40]      STag ::= '<' Name (S Attribute)* S? '>'      [WFC: Unique Att Spec ]
[41] Attribute ::= Name Eq AttValue                    [VC: Attribute Value Type ]
                                                       [WFC: No External Entity References ]
                                                       [WFC: No < in Attribute Values ]
```

The Name in the start- and end-tags gives the element's **type**. [Definition:] The Name-AttValue pairs are referred to as the **attribute specifications** of the element, [Definition:] with the Name in each pair referred to as the **attribute name** and [Definition:] the content of the AttValue (the text between the ' or " delimiters) as the **attribute value**.

Well-Formedness Constraint: Unique Att Spec
No attribute name may appear more than once in the same start-tag or empty-element tag.

Validity Constraint: Attribute Value Type
The attribute must have been declared; the value must be of the type declared for it. (For attribute types, see "3.3 Attribute-List Declarations".)

Well-Formedness Constraint: No External Entity References
Attribute values cannot contain direct or indirect entity references to external entities.

Well-Formedness Constraint: No < in Attribute Values
The replacement text of any entity referred to directly or indirectly in an attribute value (other than "<") must not contain a <.

An example of a start-tag:

```
<termdef id="dt-dog" term="dog">
```

[Definition:] The end of every element that begins with a start-tag must be marked by an **end-tag** containing a name that echoes the element's type as given in the start-tag:

End-tag

```
[42] ETag ::= '</' Name S? '>'
```

An example of an end-tag:

```
</termdef>
```

[Definition:] The text between the start-tag and end-tag is called the element's **content**:

Content of Elements

```
[43] content ::= (element | CharData
                | Reference | CDSect | PI
                | Comment)*
```

[Definition:] If an element is **empty**, it must be represented either by a start-tag immediately followed by an end-tag or by an empty-element tag. [Definition:] An **empty-element tag** takes a special form:

Tags for Empty Elements

```
[44] EmptyElemTag ::= '<' Name (S Attribute)* S? '/>'      [WFC: Unique Att Spec ]
```

Empty-element tags may be used for any element which has no content, whether or not it is declared using the keyword EMPTY. For interoperability, the empty-element tag must be used, and can only be used, for elements which are declared EMPTY.

Examples of empty elements:

```
<IMG align="left"
 src="http://www.w3.org/Icons/WWW/w3c_home" />
<br></br>
<br/>
```

3.2 Element Type Declarations

The element structure of an XML document may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element's content. (T)

Element type declarations often constrain which element types can appear as children of the element. At user option, an XML processor may issue a warning when a declaration mentions an element type for which no declaration is provided, but this is not an error. (T)

[Definition:] An **element type declaration** takes the form:

Element Type Declaration

```
[45]  elementdecl ::= '<!ELEMENT' S Name S contentspec S? '>'   [VC: Unique Element Type Declaration]
[46]  contentspec ::= 'EMPTY' | 'ANY' | Mixed | children
```

where the Name gives the element type being declared. (H)

Validity Constraint: Unique Element Type Declaration
No element type may be declared more than once. (T)

Examples of element type declarations:

```
<!ELEMENT br EMPTY>
<!ELEMENT p (#PCDATA|emph)* >
<!ELEMENT %name.para; %content.para; >
<!ELEMENT container ANY>
```

3.2.1 Element Content

[Definition:] An element type has **element content** when elements of that type must contain only child elements (no character data), optionally separated by white space (characters matching the nonterminal S). In this case, the constraint includes a content model, a simple grammar (U) governing the allowed types of the child elements and the order in which they are allowed to appear. The grammar is built on content particles (cps), which consist of names, choice lists of content particles, or sequence lists of content particles:

Element-content Models

```
[47] children ::= (choice | seq) ('?' | '*' | '+')?
[48]       cp ::= (Name | choice | seq) ('?' | '*' | '+')?
[49]   choice ::= '(' S? cp ( S? '|' S? cp )* S? ')'   [VC: Proper Group/PE Nesting ]
[50]      seq ::= '(' S? cp ( S? ',' S? cp )* S? ')'   [VC: Proper Group/PE Nesting ]
```

where each Name is the type of an element which may appear as a child. Any content particle in a choice list may appear in the element content at the location where the choice list appears in the grammar; content particles occurring in a sequence list must each appear in the element content in the order given in the list. The optional

character following a name or list governs whether the element or the content particles in the list may occur one or more (+), zero or more (*), or zero or one times (?). The absence of such an operator means that the element or content particle must appear exactly once. This syntax and meaning are identical to those used in the productions in this specification.

The content of an element matches a content model if and only if it is possible to trace out a path through the content model, obeying the sequence, choice, and repetition operators and matching each element in the content against an element type in the content model. For compatibility, it is an error if an element in the document can match more than one occurrence of an element type in the content model. For more information, see "E. Deterministic Content Models". Ⓤ

Validity Constraint: Proper Group/PE Nesting
Parameter-entity replacement text must be properly nested with parenthetized groups. That is to say, if either of the opening or closing parentheses in a `choice`, `seq`, or `Mixed` construct is contained in the replacement text for a parameter entity, both must be contained in the same replacement text. Ⓣ For interoperability, if a parameter-entity reference appears in a `choice`, `seq`, or `Mixed` construct, its replacement text should not be empty, and neither the first nor last non-blank character of the replacement text should be a connector (`|` or `,`).

Examples of element-content models:

```
<!ELEMENT spec (front, body, back?)>
<!ELEMENT div1 (head, (p | list | note)*, div2*)>
<!ELEMENT dictionary-body (%div.mix; | %dict.mix;)*>
```

3.2.2 Mixed Content

[Definition:] An element type has **mixed content** when elements of that type may contain character data, optionally interspersed with child elements. In this case, the types of the child elements may be constrained, but not their order or their number of occurrences: Ⓣ

Mixed-content Declaration

```
[51] Mixed ::= '(' S?
               '#PCDATA' Ⓤ (S?
               '|' S? Name)*
               S? ')*'
               | '(' S?          [ VC: Proper
               '#PCDATA' S?        Group/PE
               ')'                 Nesting ]
                                 [ VC: No
                                   Duplicate Types
                                   ]
```

where the `Name`s give the types of elements that may appear as children.

Validity Constraint: No Duplicate Types
The same name must not appear more than once in a single mixed-content declaration.

Examples of mixed content declarations:

```
<!ELEMENT p (#PCDATA|a|ul|b|i|em)*>
<!ELEMENT p (#PCDATA | %font; | %phrase; | %special; | %form;)* >
<!ELEMENT b (#PCDATA)>
```

3.3 Attribute-List Declarations

Attributes are used to associate name-value pairs with elements. Attribute specifications may appear only within start-tags and empty-element tags; thus, the productions used to recognize them appear in "3.1 Start-Tags, End-Tags, and Empty-Element Tags". Attribute-list declarations may be used:

- To define the set of attributes pertaining to a given element type.
- To establish type constraints for these attributes.
- To provide default values for attributes.

[Definition:] **Attribute-list declarations** specify the name, data type, and default value (if any) of each attribute associated with a given element type: Ⓣ

Attribute-list Declaration

```
[52] AttlistDecl ::= '<!ATTLIST' S Name
                     AttDef* S? '>'
[53]      AttDef ::= S Name S AttType S
                     DefaultDecl
```

The Name in the AttlistDecl rule is the type of an element. At user option, an XML processor may issue a warning if attributes are declared for an element type not itself declared, but this is not an error. The Name in the AttDef rule is the name of the attribute.

When more than one AttlistDecl is provided for a given element type, the contents of all those provided are merged. When more than one definition is provided for the same attribute of a given element type, the first declaration is binding and later declarations are ignored. For interoperability, writers of DTDs may choose to provide at most one attribute-list declaration for a given element type, at most one attribute definition for a given attribute name, and at least one attribute definition in each attribute-list declaration. For interoperability, an XML processor may at user option issue a warning when more than one attribute-list declaration is provided for a given element type, or more than one attribute definition is provided for a given attribute, but this is not an error.

3.3.1 Attribute Types

XML attribute types are of three kinds: a string type, a set of tokenized types, and enumerated types. The string type may take any literal string as a value; the tokenized types have varying lexical and semantic constraints, as noted:

Attribute Types

```
[54]       AttType ::= StringType
                     | TokenizedType
                     | EnumeratedType
[55]    StringType ::= 'CDATA'
[56] TokenizedType ::= 'ID'            [ VC: ID ]
                                       [ VC: One
                                         ID per
                                         Element
                                         Type ]
                                       [ VC: ID
                                         Attribute
                                         Default ]
                     | 'IDREF'         [ VC: IDREF
                                         ]
                     | 'IDREFS'        [ VC: IDREF
                                         ]
                     | 'ENTITY'        [ VC:
                                         Entity
                                         Name ]
                     | 'ENTITIES'      [ VC:
                                         Entity
                                         Name ]
                     | 'NMTOKEN'       [ VC: Name
                                         Token ]
                     | 'NMTOKENS'      [ VC: Name
                                         Token ]
```

Validity Constraint: ID
Values of type ID must match the Name production. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must uniquely identify the elements which bear them.

Validity Constraint: One ID per Element Type
No element type may have more than one ID attribute specified.

Validity Constraint: ID Attribute Default
An ID attribute must have a declared default of #IMPLIED or #REQUIRED.

Validity Constraint: IDREF
Values of type IDREF must match the Name production, and values of type IDREFS must match Names; each Name must match the value of an ID attribute on some element in the XML document; i.e. IDREF values must match the value of some ID attribute.

Validity Constraint: Entity Name

Values of type ENTITY must match the Name production, values of type ENTITIES must match Names; each Name must match the name of an unparsed entity declared in the DTD. Ⓣ

Validity Constraint: Name Token
Values of type NMTOKEN must match the Nmtoken production; values of type NMTOKENS must match Nmtokens.

[Definition:] **Enumerated attributes** can take one of a list of values provided in the declaration. There are two kinds of enumerated types:

Enumerated Attribute Types

```
[57] EnumeratedType ::= NotationType
                      | Enumeration
[58]   NotationType ::= 'NOTATION' S  [VC:
                        '(' S? Name    Notation
                        (S? '|' S?     Attributes
                        Name)* S? ')'  ]
[59]    Enumeration ::= '(' S?        [VC:
                        Nmtoken (S?    Enumeration
                        '|' S?         ]
                        Nmtoken)* S?
                        ')'
```

A NOTATION attribute identifies a notation, declared in the DTD with associated system and/or public identifiers, to be used in interpreting the element to which the attribute is attached. Ⓣ

Validity Constraint: Notation Attributes
Values of this type must match one of the notation names included in the declaration; all notation names in the declaration must be declared.

Validity Constraint: Enumeration
Values of this type must match one of the Nmtoken tokens in the declaration. Ⓣ

For interoperability, the same Nmtoken should not occur more than once in the enumerated attribute types of a single element type. Ⓗ

3.3.2 Attribute Defaults

An attribute declaration provides information on whether the attribute's presence is required, and if not, how an XML processor should react if a declared attribute is absent in a document.

Attribute Defaults

```
[60] DefaultDecl ::= '#REQUIRED'
                   | '#IMPLIED'
                   | (('#FIXED'   [VC:
                   S)? AttValue)   Required
                                   Attribute
                                   ]
                                  [VC:
                                   Attribute
                                   Default
                                   Legal ]
                                  [WFC: No <
                                   in
                                   Attribute
                                   Values ]
                                  [VC: Fixed
                                   Attribute
                                   Default ]
```

In an attribute declaration, #REQUIRED means that the attribute must always be provided, #IMPLIED that no default value is provided. Ⓣ [Definition:] If the declaration is neither #REQUIRED nor #IMPLIED, Ⓣ then the AttValue value contains the declared **default** value; the #FIXED keyword states that the attribute must always have the default value. Ⓣ If a default value is declared, when an XML processor encounters an omitted attribute, it is to behave as though the attribute were present with the declared default value. Ⓤ

Validity Constraint: Required Attribute
If the default declaration is the keyword #REQUIRED, then the attribute must be specified for all elements of the type in the attribute-list declaration.

Validity Constraint: Attribute Default Legal
The declared default value must meet the lexical constraints of the declared attribute type. Ⓣ

Validity Constraint: Fixed Attribute Default
If an attribute has a default value declared with the `#FIXED` keyword, instances of that attribute must match the default value.

Examples of attribute-list declarations:

```
<!ATTLIST termdef
          id      ID      #REQUIRED
          name    CDATA   #IMPLIED>
<!ATTLIST list
          type    (bullets|ordered|glossary)  "ordered">
<!ATTLIST form
          method  CDATA   #FIXED "POST">
```

3.3.3 Attribute-Value Normalization

Before the value of an attribute is passed to the application or checked for validity, the XML processor must normalize it as follows:Ⓣ

- a character reference is processed by appending the referenced character to the attribute value
- an entity reference is processed by recursively processing the replacement text of the entity
- a whitespace character (#x20, #xD, #xA, #x9) is processed by appending #x20 to the normalized value, except that only a single #x20 is appended for a "#xD#xA" sequence that is part of an external parsed entity or the literal entity value of an internal parsed entity
- other characters are processed by appending them to the normalized value

If the declared value is not CDATA, then the XML processor must further process the normalized attribute value by discarding any leading and trailing space (#x20) characters, and by replacing sequences of space (#x20) characters by a single space (#x20) character.

All attributes for which no declaration has been read should be treated by a non-validating parser as if declared `CDATA`.

3.4 Conditional Sections

[Definition:] **Conditional sections** are portions of the document type declaration external subset which are included in, or excluded from, the logical structure of the DTD based on the keyword which governs them.Ⓤ

Conditional Section

```
[61]    conditionalSect ::= includeSect
                          | ignoreSect
[62]        includeSect ::= '<![' S? 'INCLUDE'
                            S? '['
                            extSubsetDecl ']]>'
[63]         ignoreSect ::= '<![' S? 'IGNORE'
                            S? '['
                            ignoreSectContents*
                            ']]>'
[64] ignoreSectContents ::= Ignore ('<!['
                            ignoreSectContents
                            ']]>' Ignore)*
[65]             Ignore ::= Char* - (Char*
                            ('<![' | ']]>')
                            Char*)
```

Like the internal and external DTD subsets, a conditional section may contain one or more complete declarations, comments, processing instructions, or nested conditional sections, intermingled with white space.

If the keyword of the conditional section is `INCLUDE`, then the contents of the conditional section are part of the DTD. If the keyword of the conditional section is `IGNORE`, then the contents of the conditional section are not logically part of the DTD. Note that for reliable parsing, the contents of even ignored conditional sections must be read in order to detect nested conditional sections and ensure that the end of the outermost (ignored) conditional section is properly detected. If a conditional section with a keyword of `INCLUDE` occurs within a larger conditional section with a keyword of `IGNORE`, both the outer and the inner conditional sections are ignored.

If the keyword of the conditional section is a parameter-entity reference, the parameter entity must be replaced by its content before the processor decides whether to include or ignore the conditional section.Ⓣ

An example:

```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >

<![%draft;[
<!ELEMENT book (comments*, title, body, supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>
```

4. Physical Structures

[Definition:] An XML document may consist of one or many storage units. These are called **entities**; they all have **content** and are all (except for the document entity, see below, and the external DTD subset) identified by **name**. Each XML document has one entity called the document entity, which serves as the starting point for the XML processor and may contain the whole document.Ⓣ

Entities may be either parsed or unparsed.Ⓤ [Definition:] A **parsed entity's** contents are referred to as its replacement text; this text is considered an integral part of the document.

[Definition:] An **unparsed entity** is a resource whose contents may or may not be text, and if text, may not be XML. Each unparsed entity has an associated notation, identified by name. Beyond a requirement that an XML processor make the identifiers for the entity and notation available to the application, XML places no constraints on the contents of unparsed entities.

Parsed entities are invoked by name using entity references; unparsed entities by name, given in the value of `ENTITY` or `ENTITIES` attributes.

[Definition:] **General entities** are entities for use within the document content. In this specification, general entities are sometimes referred to with the unqualified term *entity* when this leads to no ambiguity. [Definition:] Parameter entities are parsed entities for use within the DTD. These two types of entities use different forms of reference and are recognized in different contexts. Furthermore, they occupy different namespaces; a parameter entity and a general entity with the same name are two distinct entities. Ⓣ

4.1 Character and Entity References

[Definition:] A **character reference** refers to a specific character in the ISO/IEC 10646 character set, for example one not directly accessible from available input devices.Ⓣ

Character Reference

```
[66] CharRef ::= '&#' [0-9]+ ';'
                 | '&#x' [0-9a-fA-F]+ ';'   [WFC: Legal Character ]
```

Well-Formedness Constraint: Legal Character
Characters referred to using character references must match the production for Char.

If the character reference begins with "`&#x`", the digits and letters up to the terminating `;` provide a hexadecimal representation of the character's code point in ISO/IEC 10646.Ⓣ If it begins just with "`&#`", the digits up to the terminating `;` provide a decimal representation of the character's code point.

[Definition:] An **entity reference** refers to the content of a named entity. [Definition:] References to parsed general entities use ampersand (`&`) and semicolon (`;`) as delimiters.Ⓤ [Definition:] **Parameter-entity references** use percent-sign (`%`) and semicolon (`;`) as delimiters.

Entity Reference

```
[67]   Reference ::= EntityRef
                   | CharRef
[68]   EntityRef ::= '&' Name ';'  [WFC: Entity Declared ]
                                   [VC: Entity Declared ]
                                   [WFC: Parsed Entity ]
                                   [WFC: No Recursion ]
[69] PEReference ::= '%' Name ';'  [VC: Entity Declared ]
```

```
                                  [WFC: No
                                   Recursion ]
                                  [WFC: In DTD
                                   ]
```

Well-Formedness Constraint: Entity Declared
In a document without any DTD, a document with only an internal DTD subset which contains no parameter entity references, or a document with "`standalone='yes'`", the `Name` given in the entity reference must match that in an entity declaration, except that well-formed documents need not declare any of the following entities: `amp`, `lt`, `gt`, `apos`, `quot`. The declaration of a parameter entity must precede any reference to it. Similarly, the declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration. Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be declared is a well-formedness constraint only if standalone='yes'.

Validity Constraint: Entity Declared
In a document with an external subset or external parameter entities with "`standalone='no'`", the `Name` given in the entity reference must match that in an entity declaration. For interoperability, valid documents should declare the entities `amp`, `lt`, `gt`, `apos`, `quot`, in the form specified in "4.6 Predefined Entities". The declaration of a parameter entity must precede any reference to it. Similarly, the declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration.

Well-Formedness Constraint: Parsed Entity
An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type `ENTITY` or `ENTITIES`.

Well-Formedness Constraint: No Recursion
A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

Well-Formedness Constraint: In DTD
Parameter-entity references may only appear in the DTD.

Examples of character and entity references:

```
Type <key>less-than</key> (&#x3C;) to save options.
This document was prepared on &docdate; and
is classified &security-level;.
```

Example of a parameter-entity reference:

```
<!-- declare the parameter entity "ISOLat2"... -->
<!ENTITY % ISOLat2
         SYSTEM "http://www.xml.com/iso/isolat2-xml.entities" >
<!-- ... now reference it. -->
%ISOLat2;
```

4.2 Entity Declarations

[Definition:] Entities are declared thus:

Entity Declaration

```
[70] EntityDecl ::= GEDecl | PEDecl
[71]     GEDecl ::= '<!ENTITY' S Name S
                    EntityDef S? '>'
[72]     PEDecl ::= '<!ENTITY' S '%' S Name S
                    PEDef S? '>'
[73]  EntityDef ::= EntityValue | (ExternalID
                    NDataDecl?)
[74]      PEDef ::= EntityValue | ExternalID
```

The `Name` identifies the entity in an entity reference or, in the case of an unparsed entity, in the value of an `ENTITY` or `ENTITIES` attribute. If the same entity is declared more than once, the first declaration encountered is binding; at user option, an XML processor may issue a warning if entities are declared multiple times.

4.2.1 Internal Entities

[Definition:] If the entity definition is an `EntityValue`, the defined entity is called an **internal entity**. There is no separate physical storage object, and the content of the entity is given in the declaration. Note that some processing of entity and character references in the literal entity value may be required to produce the correct

replacement text: see "4.5 Construction of Internal Entity Replacement Text".

An internal entity is a parsed entity.

Example of an internal entity declaration:

```
<!ENTITY Pub-Status "This is a pre-release of the
 specification.">
```

4.2.2 External Entities

[Definition:] If the entity is not internal, it is an **external entity**,(U) declared as follows:

External Entity Declaration		
[75] ExternalID	::=	'SYSTEM' S SystemLiteral | 'PUBLIC' S PubidLiteral S SystemLiteral
[76] NDataDecl	::=	S 'NDATA'(T) S Name [VC: Notation Declared]

If the NDataDecl is present, this is a general unparsed entity; otherwise it is a parsed entity.

Validity Constraint: Notation Declared
The Name must match the declared name of a notation. (T)

[Definition:] The SystemLiteral is called the entity's **system identifier**. It is a URI, which may be used to retrieve the entity.(T) Note that the hash mark (#) and fragment identifier frequently used with URIs are not, formally, part of the URI itself; an XML processor may signal an error if a fragment identifier is given as part of a system identifier. (T)Unless otherwise provided by information outside the scope of this specification (e.g. a special XML element type defined by a particular DTD, or a processing instruction defined by a particular application specification), relative URIs are relative to the location of the resource within which the entity declaration occurs. A URI might thus be relative to the document entity, to the entity containing the external DTD subset, or to some other external parameter entity.

An XML processor should handle a non-ASCII character in a URI by(T) representing the character in UTF-8 as one or more bytes, and then escaping these bytes with the URI escaping mechanism (i.e., by converting each byte to %HH, where HH is the hexadecimal notation of the byte value).

[Definition:] In addition to a system identifier, an external identifier may include a **public identifier**. An XML processor attempting to retrieve the entity's content may use the public identifier to try to generate an alternative URI. If the processor is unable to do so, it must use the URI specified in the system literal. (T) Before a match is attempted, all strings of white space in the public identifier must be normalized to single space characters (#x20), and leading and trailing white space must be removed.

Examples of external entity declarations:

```
<!ENTITY open-hatch
         SYSTEM "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY open-hatch
         PUBLIC "-//Textuality//TEXT Standard open-hatch boilerplate//EN"
         "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY hatch-pic
         SYSTEM "../grafix/OpenHatch.gif"
         NDATA gif >
```

4.3 Parsed Entities

4.3.1 The Text Declaration

External parsed entities may each begin with a **text declaration**.(T)

Text Declaration		
[77] TextDecl	::=	'<?xml' VersionInfo? EncodingDecl S? '?>'(T)

The text declaration must be provided literally, not by reference to a parsed entity. No text declaration may appear at

any position other than the beginning of an external parsed entity.

4.3.2 Well-Formed Parsed Entities

The document entity is well-formed if it matches the production labeled `document`. An external general parsed entity is well-formed if it matches the production labeled `extParsedEnt`. An external parameter entity is well-formed if it matches the production labeled `extPE`.

Well-Formed External Parsed Entity

```
[78] extParsedEnt ::= TextDecl? content
[79]        extPE ::= TextDecl? extSubsetDecl
```

An internal general parsed entity is well-formed if its replacement text matches the production labeled `content`. All internal parameter entities are well-formed by definition.

A consequence of well-formedness in entities is that the logical and physical structures in an XML document are properly nested; no start-tag, end-tag, empty-element tag, element, comment, processing instruction, character reference, or entity reference can begin in one entity and end in another.

4.3.3 Character Encoding in Entities

Each external parsed entity in an XML document may use a different encoding for its characters. All XML processors must be able to read entities in either UTF-8 or UTF-16.

Entities encoded in UTF-16 must begin with the Byte Order Mark described by ISO/IEC 10646 Annex E and Unicode Appendix B (the ZERO WIDTH NO-BREAK SPACE character, #xFEFF). This is an encoding signature, not part of either the markup or the character data of the XML document. XML processors must be able to use this character to differentiate between UTF-8 and UTF-16 encoded documents.

Although an XML processor is required to read only entities in the UTF-8 and UTF-16 encodings, it is recognized that other encodings are used around the world, and it may be desired for XML processors to read entities that use them. Parsed entities which are stored in an encoding other than UTF-8 or UTF-16 must begin with a text declaration containing an encoding declaration:

Encoding Declaration

```
[80] EncodingDecl ::= S 'encoding'
                      Eq ('"'
                      EncName '"' |
                      "'" EncName
                      "'" )
[81]      EncName ::= [A-Za-z]      /* Encoding
                      ([A-Za-z0-9._]   name
                      | '-')*          contains
                                       only Latin
                                       characters
                                       */
```

In the document entity, the encoding declaration is part of the XML declaration. The `EncName` is the name of the encoding used.

In an encoding declaration, the values "`UTF-8`", "`UTF-16`", "`ISO-10646-UCS-2`", and "`ISO-10646-UCS-4`" should be used for the various encodings and transformations of Unicode / ISO/IEC 10646, the values "`ISO-8859-1`", "`ISO-8859-2`", ... "`ISO-8859-9`" should be used for the parts of ISO 8859, and the values "`ISO-2022-JP`", "`Shift_JIS`", and "`EUC-JP`" should be used for the various encoded forms of JIS X-0208-1997. XML processors may recognize other encodings; it is recommended that character encodings registered (as *charsets*) with the Internet Assigned Numbers Authority [IANA], other than those just listed, should be referred to using their registered names. Note that these registered names are defined to be case-insensitive, so processors wishing to match against them should do so in a case-insensitive way.

In the absence of information provided by an external transport protocol (e.g. HTTP or MIME), it is an error for an entity including an encoding declaration to be presented to the XML processor in an encoding other than that named in the declaration, for an encoding declaration to occur other than at the beginning of an external entity, or for an entity which begins with neither a Byte Order Mark nor an encoding declaration to use an encoding other than UTF-8. Note that since ASCII is a subset of UTF-8, ordinary ASCII entities do not strictly need an encoding declaration.

It is a fatal error when an XML processor encounters an entity with an encoding that it is unable to process.

Examples of encoding declarations:

```
<?xml encoding='UTF-8'?>
<?xml encoding='EUC-JP'?>
```

4.4 XML Processor Treatment of Entities and References

The table below summarizes the contexts in which character references, entity references, and invocations of unparsed entities might appear and the required behavior of an XML processor in each case. The labels in the leftmost column describe the recognition context:

Reference in Content

as a reference anywhere after the start-tag and before the end-tag of an element; corresponds to the nonterminal `content`.

Reference in Attribute Value

as a reference within either the value of an attribute in a start-tag, or a default value in an attribute declaration; corresponds to the nonterminal `AttValue`.

Occurs as Attribute Value

as a `Name`, not a reference, appearing either as the value of an attribute which has been declared as type `ENTITY`, or as one of the space-separated tokens in the value of an attribute which has been declared as type `ENTITIES`.

Reference in Entity Value

as a reference within a parameter or internal entity's literal entity value in the entity's declaration; corresponds to the nonterminal `EntityValue`.

Reference in DTD

as a reference within either the internal or external subsets of the DTD, but outside of an `EntityValue` or `AttValue`.

	Entity Type				Character
	Parameter	Internal General	External Parsed General	Unparsed	
Reference in Content	Not recognized	Included	Included if validating	Forbidden	Included
Reference in Attribute Value	Not recognized	Included in literal	Forbidden	Forbidden	Included
Occurs as Attribute Value	Not recognized	Forbidden	Forbidden	Notify	Not recognized
Reference in EntityValue	Included in literal	Bypassed	Bypassed	Forbidden	Included
Reference in DTD	Included as PE	Forbidden	Forbidden	Forbidden	Forbidden

4.4.1 Not Recognized

Outside the DTD, the % character has no special significance; thus, what would be parameter entity references in the DTD are not recognized as markup in `content`. Similarly, the names of unparsed entities are not recognized except when they appear in the value of an appropriately declared attribute.

4.4.2 Included

[Definition:] An entity is **included** when its replacement text is retrieved and processed, in place of the reference itself, as though it were part of the document at the location the reference was recognized. The replacement text may contain both character data and (except for parameter entities) markup, which must be recognized in the usual way, except that the replacement text of entities used to escape markup delimiters (the entities `amp`, `lt`, `gt`, `apos`, `quot`) is always treated as data. (The string "`AT&T;`" expands to "`AT&T;`" and the remaining ampersand is not recognized as an entity-reference delimiter.) A character reference is **included** when the indicated character is processed in place of the reference itself.

4.4.3 Included If Validating

When an XML processor recognizes a reference to a parsed entity, in order to validate the document, the processor must include its replacement text. If the entity is external, and the processor is not attempting to validate the XML document, the processor may, but need not, include the entity's replacement text. If a non-validating parser does not include the replacement text, it must inform the application that it recognized, but did not read, the entity.

This rule is based on the recognition that the automatic inclusion provided by the SGML and XML entity mechanism, primarily designed to support modularity in authoring, is not necessarily appropriate for other

applications, in particular document browsing. (H) Browsers, for example, when encountering an external parsed entity reference, might choose to provide a visual indication of the entity's presence and retrieve it for display only on demand.

4.4.4 Forbidden

The following are forbidden, and constitute fatal errors:

- the appearance of a reference to an unparsed entity.
- the appearance of any character or general-entity reference in the DTD except within an `EntityValue` or `AttValue`. (H)
- a reference to an external entity in an attribute value. (H)

4.4.5 Included in Literal

When an entity reference appears in an attribute value, or a parameter entity reference appears in a literal entity value, its replacement text is processed in place of the reference itself as though it were part of the document at the location the reference was recognized, except that a single or double quote character in the replacement text is always treated as a normal data character and will not terminate the literal. For example, this is well-formed: (T)

```
<!ENTITY % YN '"Yes"' >
<!ENTITY WhatHeSaid "He said &YN;" >
```

while this is not:

```
<!ENTITY EndAttr "27'" >
<element attribute='a-&EndAttr;>
```

4.4.6 Notify

When the name of an unparsed entity appears as a token in the value of an attribute of declared type `ENTITY` or `ENTITIES`, a validating processor must inform the application of the system and public (if any) identifiers for both the entity and its associated notation. (T)

4.4.7 Bypassed

When a general entity reference appears in the `EntityValue` in an entity declaration, it is bypassed and left as is. (T)

4.4.8 Included as PE

Just as with external parsed entities, parameter entities need only be included if validating. When a parameter-entity reference is recognized in the DTD and included, its replacement text is enlarged by the attachment of one leading and one following space (#x20) character; the intent is to constrain the replacement text of parameter entities to contain an integral number of grammatical tokens in the DTD. (U)

4.5 Construction of Internal Entity Replacement Text

In discussing the treatment of internal entities, it is useful to distinguish two forms of the entity's value. [Definition:] The **literal entity value** is the quoted string actually present in the entity declaration, corresponding to the non-terminal `EntityValue`. [Definition:] The **replacement text** is the content of the entity, after replacement of character references and parameter-entity references.

The literal entity value as given in an internal entity declaration (`EntityValue`) may contain character, parameter-entity, and general-entity references. Such references must be contained entirely within the literal entity value. The actual replacement text that is included as described above must contain the *replacement text* of any parameter entities referred to, and must contain the character referred to, in place of any character references in the literal entity value; however, general-entity references must be left as-is, unexpanded. For example, given the following declarations:

```
<!ENTITY % pub    "&#xc9;ditions Gallimard" >
<!ENTITY   rights "All rights reserved" >
<!ENTITY   book   "La Peste: Albert Camus,
&#xA9; 1947 %pub;. &rights;" >(T)
```

then the replacement text for the entity "`book`" is:

```
La Peste: Albert Camus,
© 1947 Éditions Gallimard. &rights;
```

The general-entity reference "`&rights;`" would be expanded should the reference "`&book;`" appear in the document's content or an attribute value.

These simple rules may have complex interactions; for a detailed discussion of a difficult example, see "D. Expansion of Entity and Character References".

4.6 Predefined Entities

[Definition:] Entity and character references can both be used to **escape** the left angle bracket, ampersand, and other delimiters. A set of general entities (`amp`, `lt`, `gt`, `apos`, `quot`) is specified for this purpose. Numeric character references may also be used; they are expanded immediately when recognized and must be treated as character data, so the numeric character references "`<`" and "`&`" may be used to escape < and & when they occur in character data.Ⓣ

All XML processors must recognize these entities whether they are declared or not. For interoperability, valid XML documents should declare these entities, like any others, before using them. If the entities in question are declared, they must be declared as internal entities whose replacement text is the single character being escaped or a character reference to that character, as shown below.

```
<!ENTITY lt     "&#38;#60;">
<!ENTITY gt     "&#62;">
<!ENTITY amp    "&#38;#38;">
<!ENTITY apos   "&#39;">
<!ENTITY quot   "&#34;">
```

Note that the < and & characters in the declarations of "`lt`" and "`amp`" are doubly escaped to meet the requirement that entity replacement be well-formed.

4.7 Notation Declarations

[Definition:] **Notations** identify by name the format of unparsed entities, the format of elements which bear a notation attribute, or the application to which a processing instruction is addressed.

[Definition:] **Notation declarations** provide a name for the notation, for use in entity and attribute-list declarations and in attribute specifications, and an external identifier for the notation which may allow an XML processor or its client application to locate a helper application capable of processing data in the given notation.Ⓣ

Notation Declarations

```
[82] NotationDecl ::= '<!NOTATION'Ⓤ S Name S
                      (ExternalID |  PublicID)
                      S? '>'
[83]     PublicID ::= 'PUBLIC' S PubidLiteral
```

XML processors must provide applications with the name and external identifier(s) of any notation declared and referred to in an attribute value, attribute definition, or entity declaration. They may additionally resolve the external identifier into the system identifier, file name, or other information needed to allow the application to call a processor for data in the notation described.Ⓤ (It is not an error, however, for XML documents to declare and refer to notations for which notation-specific applications are not available on the system where the XML processor or application is running.)

4.8 Document Entity

[Definition:] The **document entity** serves as the root of the entity tree and a starting-point for an XML processor.Ⓣ This specification does not specify how the document entity is to be located by an XML processor; unlike other entities, the document entity has no name and might well appear on a processor input stream without any identification at all.Ⓔ

5. Conformance

5.1 Validating and Non-Validating Processors

Conforming XML processors fall into two classes: validating and non-validating.Ⓤ

Validating and non-validating processors alike must report violations of this specification's well-formedness constraints in the content of the document entity and any other parsed entities that they read.

[Definition:] **Validating processors** must report violations of the constraints expressed by the declarations in the DTD, and failures to fulfill the validity constraints given in this specification. To accomplish this, validating XML processors must read and process the entire DTD and all external parsed entities referenced in the document.

Non-validating processors are required to check only the document entity, including the entire internal DTD subset, for well-formedness. [Definition:] Ⓣ While they are not required to check the document for validity, they are required to **process** all the declarations they read in the internal DTD subset and in any parameter entity that they read, up to the first reference to a parameter entity that they do *not* read; that is to say, they must use the information

in those declarations to normalize attribute values, include the replacement text of internal entities, and supply default attribute values. (U) They must not process entity declarations or attribute-list declarations encountered after a reference to a parameter entity that is not read, since the entity may have contained overriding declarations. (E)

5.2 Using XML Processors

The behavior of a validating XML processor is highly predictable; it must read every piece of a document and report all well-formedness and validity violations. Less is required of a non-validating processor; it need not read any part of the document other than the document entity.(U) This has two effects that may be important to users of XML processors:

- Certain well-formedness errors, specifically those that require reading external entities, may not be detected by a non-validating processor. Examples include the constraints entitled Entity Declared, Parsed Entity, and No Recursion, as well as some of the cases described as forbidden in "4.4 XML Processor Treatment of Entities and References".
- The information passed from the processor to the application may vary, depending on whether the processor reads parameter and external entities. For example, a non-validating processor may not normalize attribute values, include the replacement text of internal entities, or supply default attribute values, where doing so depends on having read declarations in external or parameter entities.

For maximum reliability in interoperating between different XML processors, applications which use non-validating processors should not rely on any behaviors not required of such processors. Applications which require facilities such as the use of default attributes or internal entities which are declared in external entities should use validating XML processors.

6. Notation

The formal grammar of XML is given in this specification using a simple Extended Backus-Naur Form (EBNF) notation. Each rule in the grammar defines one symbol, in the form

```
symbol ::= expression
```

Symbols are written with an initial capital letter if they are defined by a regular expression, or with an initial lower case letter otherwise. Literal strings are quoted.

Within the expression on the right-hand side of a rule, the following expressions are used to match strings of one or more characters:

#xN

where N is a hexadecimal integer, the expression matches the character in ISO/IEC 10646 whose canonical (UCS-4) code value, when interpreted as an unsigned binary number, has the value indicated. The number of leading zeros in the #xN form is insignificant; the number of leading zeros in the corresponding code value is governed by the character encoding in use and is not significant for XML.

[a-zA-Z], [#xN-#xN]

matches any character with a value in the range(s) indicated (inclusive).

[^a-z], [^#xN-#xN]

matches any character with a value *outside* the range indicated.

[^abc], [^#xN#xN#xN]

matches any character with a value not among the characters given.

"string"

matches a literal string matching that given inside the double quotes.

'string'

matches a literal string matching that given inside the single quotes.

These symbols may be combined to match more complex patterns as follows, where A and B represent simple expressions:

(expression)

expression is treated as a unit and may be combined as described in this list.

A?

matches A or nothing; optional A.

A B

matches A followed by B.

A | B

matches A or B but not both.

A - B

matches any string that matches A but does not match B.

A+

matches one or more occurrences of A.

A*

matches zero or more occurrences of A.

Other notations used in the productions are:

/* ... */

comment.

[wfc: ...]

well-formedness constraint; this identifies by name a constraint on well-formed documents associated with a production.

[vc: ...]

validity constraint; this identifies by name a constraint on valid documents associated with a production.

Appendices

A. References

A.1 Normative References

IANA

(Internet Assigned Numbers Authority) *Official Names for Character Sets*, ed. Keld Simonsen et al. See ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

IETF RFC 1766

IETF (Internet Engineering Task Force). *RFC 1766: Tags for the Identification of Languages*, ed. H. Alvestrand. 1995.

ISO 639

(International Organization for Standardization). *ISO 639:1988 (E). Code for the representation of names of languages.* [Geneva]: International Organization for Standardization, 1988.

ISO 3166

(International Organization for Standardization). *ISO 3166-1:1997 (E). Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes* [Geneva]: International Organization for Standardization, 1997.

ISO/IEC 10646

ISO (International Organization for Standardization). *ISO/IEC 10646-1993 (E). Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane.* [Geneva]: International Organization for Standardization, 1993 (plus amendments AM 1 through AM 7).

Unicode

The Unicode Consortium. *The Unicode Standard, Version 2.0.* Reading, Mass.: Addison-Wesley Developers Press, 1996.

A.2 Other References

Aho/Ullman

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading: Addison-Wesley, 1986, rpt. corr. 1988.

Berners-Lee et al.

Berners-Lee, T., R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax and Semantics*. 1997. (Work in progress; see updates to RFC1738.)

Brüggemann-Klein

Brüggemann-Klein, Anne. *Regular Expressions into Finite Automata*. Extended abstract in I. Simon, Hrsg., LATIN 1992, S. 97-98. Springer-Verlag, Berlin 1992. Full Version in Theoretical Computer Science 120: 197-213, 1993.

Brüggemann-Klein and Wood

Brüggemann-Klein, Anne, and Derick Wood. *Deterministic Regular Languages*. Universität Freiburg, Institut für Informatik, Bericht 38, Oktober 1991.

Clark

James Clark. Comparison of SGML and XML. See http://www.w3.org/TR/NOTE-sgml-xml-971215.

IETF RFC1738

IETF (Internet Engineering Task Force). *RFC 1738: Uniform Resource Locators (URL)*, ed. T. Berners-Lee, L. Masinter, M. McCahill. 1994.

IETF RFC1808

IETF (Internet Engineering Task Force). *RFC 1808: Relative Uniform Resource Locators*, ed. R. Fielding.

1995.

IETF RFC2141

IETF (Internet Engineering Task Force). *RFC 2141: URN Syntax*, ed. R. Moats. 1997.

ISO 8879

ISO (International Organization for Standardization). *ISO 8879:1986(E). Information processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML).* First edition -- 1986-10-15. [Geneva]: International Organization for Standardization, 1986.

ISO/IEC 10744

ISO (International Organization for Standardization). *ISO/IEC 10744-1992 (E). Information technology -- Hypermedia/Time-based Structuring Language (HyTime).* [Geneva]: International Organization for Standardization, 1992. *Extended Facilities Annexe.* [Geneva]: International Organization for Standardization, 1996.

B. Character Classes

Following the characteristics defined in the Unicode standard, characters are classed as base characters (among others, these contain the alphabetic characters of the Latin alphabet, without diacritics), ideographic characters, and combining characters (among others, this class contains most diacritics); these classes combine to form the class of letters. Digits and extenders are also distinguished.

Characters

```
[84]        Letter ::= BaseChar | Ideographic
[85]      BaseChar ::= [#x0041-#x005A]
                     | [#x0061-#x007A]
                     | [#x00C0-#x00D6]
                     | [#x00D8-#x00F6]
                     | [#x00F8-#x00FF]
                     | [#x0100-#x0131]
                     | [#x0134-#x013E]
                     | [#x0141-#x0148]
                     | [#x014A-#x017E]
                     | [#x0180-#x01C3]
                     | [#x01CD-#x01F0]
                     | [#x01F4-#x01F5]
                     | [#x01FA-#x0217]
                     | [#x0250-#x02A8]
                     | [#x02BB-#x02C1]
                     | #x0386
                     | [#x0388-#x038A]
                     | #x038C
                     | [#x038E-#x03A1]
                     | [#x03A3-#x03CE]
                     | [#x03D0-#x03D6]
                     | #x03DA | #x03DC
                     | #x03DE | #x03E0
                     | [#x03E2-#x03F3]
                     | [#x0401-#x040C]
                     | [#x040E-#x044F]
                     | [#x0451-#x045C]
                     | [#x045E-#x0481]
                     | [#x0490-#x04C4]
                     | [#x04C7-#x04C8]
                     | [#x04CB-#x04CC]
                     | [#x04D0-#x04EB]
                     | [#x04EE-#x04F5]
                     | [#x04F8-#x04F9]
                     | [#x0531-#x0556]
                     | #x0559
                     | [#x0561-#x0586]
                     | [#x05D0-#x05EA]
                     | [#x05F0-#x05F2]
                     | [#x0621-#x063A]
                     | [#x0641-#x064A]
                     | [#x0671-#x06B7]
                     | [#x06BA-#x06BE]
                     | [#x06C0-#x06CE]
                     | [#x06D0-#x06D3]
                     | #x06D5
```

```
                    | [#x06E5-#x06E6]Ⓤ
                    | [#x0905-#x0939]
                    | #x093D
                    | [#x0958-#x0961]Ⓤ
                    | [#x0985-#x098C]
                    | [#x098F-#x0990]
                    | [#x0993-#x09A8]
                    | [#x09AA-#x09B0]
                    | #x09B2
                    | [#x09B6-#x09B9]
                    | [#x09DC-#x09DD]
                    | [#x09DF-#x09E1]
                    | [#x09F0-#x09F1]Ⓤ
                    | [#x0A05-#x0A0A]
                    | [#x0A0F-#x0A10]
                    | [#x0A13-#x0A28]
                    | [#x0A2A-#x0A30]
                    | [#x0A32-#x0A33]
                    | [#x0A35-#x0A36]
                    | [#x0A38-#x0A39]
                    | [#x0A59-#x0A5C]
                    | #x0A5E
                    | [#x0A72-#x0A74]Ⓤ
                    | [#x0A85-#x0A8B]
                    | #x0A8D
                    | [#x0A8F-#x0A91]
                    | [#x0A93-#x0AA8]
                    | [#x0AAA-#x0AB0]
                    | [#x0AB2-#x0AB3]
                    | [#x0AB5-#x0AB9]
                    | #x0ABD | #x0AE0Ⓤ
                    | [#x0B05-#x0B0C]
                    | [#x0B0F-#x0B10]
                    | [#x0B13-#x0B28]
                    | [#x0B2A-#x0B30]
                    | [#x0B32-#x0B33]
                    | [#x0B36-#x0B39]
                    | #x0B3D
                    | [#x0B5C-#x0B5D]
                    | [#x0B5F-#x0B61]Ⓤ
                    | [#x0B85-#x0B8A]
                    | [#x0B8E-#x0B90]
                    | [#x0B92-#x0B95]
                    | [#x0B99-#x0B9A]
                    | #x0B9C
                    | [#x0B9E-#x0B9F]
                    | [#x0BA3-#x0BA4]
                    | [#x0BA8-#x0BAA]
                    | [#x0BAE-#x0BB5]
                    | [#x0BB7-#x0BB9]Ⓤ
                    | [#x0C05-#x0C0C]
                    | [#x0C0E-#x0C10]
                    | [#x0C12-#x0C28]
                    | [#x0C2A-#x0C33]
                    | [#x0C35-#x0C39]
                    | [#x0C60-#x0C61]Ⓤ
                    | [#x0C85-#x0C8C]
                    | [#x0C8E-#x0C90]
                    | [#x0C92-#x0CA8]
                    | [#x0CAA-#x0CB3]
                    | [#x0CB5-#x0CB9]
                    | #x0CDE
                    | [#x0CE0-#x0CE1]Ⓤ
                    | [#x0D05-#x0D0C]
                    | [#x0D0E-#x0D10]
                    | [#x0D12-#x0D28]
                    | [#x0D2A-#x0D39]
                    | [#x0D60-#x0D61]Ⓤ
                    | [#x0E01-#x0E2E]
                    | #x0E30
                    | [#x0E32-#x0E33]
```

```
                    | [#x0E40-#x0E45]Ⓤ
                    | [#x0E81-#x0E82]
                    | #x0E84
                    | [#x0E87-#x0E88]
                    | #x0E8A | #x0E8D
                    | [#x0E94-#x0E97]
                    | [#x0E99-#x0E9F]
                    | [#x0EA1-#x0EA3]
                    | #x0EA5 | #x0EA7
                    | [#x0EAA-#x0EAB]
                    | [#x0EAD-#x0EAE]
                    | #x0EB0
                    | [#x0EB2-#x0EB3]
                    | #x0EBD
                    | [#x0EC0-#x0EC4]Ⓤ
                    | [#x0F40-#x0F47]
                    | [#x0F49-#x0F69]Ⓤ
                    | [#x10A0-#x10C5]
                    | [#x10D0-#x10F6]Ⓤ
                    | #x1100
                    | [#x1102-#x1103]
                    | [#x1105-#x1107]
                    | #x1109
                    | [#x110B-#x110C]
                    | [#x110E-#x1112]
                    | #x113C | #x113E
                    | #x1140 | #x114C
                    | #x114E | #x1150
                    | [#x1154-#x1155]
                    | #x1159
                    | [#x115F-#x1161]
                    | #x1163 | #x1165
                    | #x1167 | #x1169
                    | [#x116D-#x116E]
                    | [#x1172-#x1173]
                    | #x1175 | #x119E
                    | #x11A8 | #x11AB
                    | [#x11AE-#x11AF]
                    | [#x11B7-#x11B8]
                    | #x11BA
                    | [#x11BC-#x11C2]
                    | #x11EB | #x11F0
                    | #x11F9Ⓤ
                    | [#x1E00-#x1E9B]
                    | [#x1EA0-#x1EF9]Ⓤ
                    | [#x1F00-#x1F15]
                    | [#x1F18-#x1F1D]
                    | [#x1F20-#x1F45]
                    | [#x1F48-#x1F4D]
                    | [#x1F50-#x1F57]
                    | #x1F59 | #x1F5B
                    | #x1F5D
                    | [#x1F5F-#x1F7D]
                    | [#x1F80-#x1FB4]
                    | [#x1FB6-#x1FBC]
                    | #x1FBE
                    | [#x1FC2-#x1FC4]
                    | [#x1FC6-#x1FCC]
                    | [#x1FD0-#x1FD3]
                    | [#x1FD6-#x1FDB]
                    | [#x1FE0-#x1FEC]
                    | [#x1FF2-#x1FF4]
                    | [#x1FF6-#x1FFC]Ⓤ
                    | #x2126
                    | [#x212A-#x212B]
                    | #x212EⓊ
                    | [#x2180-#x2182]Ⓤ
                    | [#x3041-#x3094]Ⓤ
                    | [#x30A1-#x30FA]Ⓤ
                    | [#x3105-#x312C]Ⓤ
                    | [#xAC00-#xD7A3]Ⓤ
```

```
[86]   Ideographic ::= [#x4E00-#x9FA5]Ⓤ
                     | #x3007
                     | [#x3021-#x3029]Ⓤ
[87] CombiningChar ::= [#x0300-#x0345]
                     | [#x0360-#x0361]Ⓤ
                     | [#x0483-#x0486]Ⓤ
                     | [#x0591-#x05A1]
                     | [#x05A3-#x05B9]
                     | [#x05BB-#x05BD]
                     | #x05BF
                     | [#x05C1-#x05C2]
                     | #x05C4Ⓤ
                     | [#x064B-#x0652]
                     | #x0670
                     | [#x06D6-#x06DC]
                     | [#x06DD-#x06DF]
                     | [#x06E0-#x06E4]
                     | [#x06E7-#x06E8]
                     | [#x06EA-#x06ED]Ⓤ
                     | [#x0901-#x0903]
                     | #x093C
                     | [#x093E-#x094C]
                     | #x094D
                     | [#x0951-#x0954]
                     | [#x0962-#x0963]Ⓤ
                     | [#x0981-#x0983]
                     | #x09BC | #x09BE
                     | #x09BF
                     | [#x09C0-#x09C4]
                     | [#x09C7-#x09C8]
                     | [#x09CB-#x09CD]
                     | #x09D7
                     | [#x09E2-#x09E3]Ⓤ
                     | #x0A02 | #x0A3C
                     | #x0A3E | #x0A3F
                     | [#x0A40-#x0A42]
                     | [#x0A47-#x0A48]
                     | [#x0A4B-#x0A4D]
                     | [#x0A70-#x0A71]Ⓤ
                     | [#x0A81-#x0A83]
                     | #x0ABC
                     | [#x0ABE-#x0AC5]
                     | [#x0AC7-#x0AC9]
                     | [#x0ACB-#x0ACD]Ⓤ
                     | [#x0B01-#x0B03]
                     | #x0B3C
                     | [#x0B3E-#x0B43]
                     | [#x0B47-#x0B48]
                     | [#x0B4B-#x0B4D]
                     | [#x0B56-#x0B57]Ⓤ
                     | [#x0B82-#x0B83]
                     | [#x0BBE-#x0BC2]
                     | [#x0BC6-#x0BC8]
                     | [#x0BCA-#x0BCD]
                     | #x0BD7Ⓤ
                     | [#x0C01-#x0C03]
                     | [#x0C3E-#x0C44]
                     | [#x0C46-#x0C48]
                     | [#x0C4A-#x0C4D]
                     | [#x0C55-#x0C56]Ⓤ
                     | [#x0C82-#x0C83]
                     | [#x0CBE-#x0CC4]
                     | [#x0CC6-#x0CC8]
                     | [#x0CCA-#x0CCD]
                     | [#x0CD5-#x0CD6]Ⓤ
                     | [#x0D02-#x0D03]
                     | [#x0D3E-#x0D43]
                     | [#x0D46-#x0D48]
                     | [#x0D4A-#x0D4D]
                     | #x0D57Ⓤ | #x0E31
```

```
                    | [#x0E34-#x0E3A]
                    | [#x0E47-#x0E4E]Ⓤ
                    | #x0EB1
                    | [#x0EB4-#x0EB9]
                    | [#x0EBB-#x0EBC]
                    | [#x0EC8-#x0ECD]Ⓤ
                    | [#x0F18-#x0F19]
                    | #x0F35 | #x0F37
                    | #x0F39 | #x0F3E
                    | #x0F3F
                    | [#x0F71-#x0F84]
                    | [#x0F86-#x0F8B]
                    | [#x0F90-#x0F95]
                    | #x0F97
                    | [#x0F99-#x0FAD]
                    | [#x0FB1-#x0FB7]
                    | #x0FB9Ⓤ
                    | [#x20D0-#x20DC]
                    | #x20E1Ⓤ
                    | [#x302A-#x302F]Ⓤ
                    | #x3099 | #x309AⓊ
[88]          Digit ::= [#x0030-#x0039]Ⓤ
                    | [#x0660-#x0669]Ⓤ
                    | [#x06F0-#x06F9]Ⓤ
                    | [#x0966-#x096F]Ⓤ
                    | [#x09E6-#x09EF]Ⓤ
                    | [#x0A66-#x0A6F]Ⓤ
                    | [#x0AE6-#x0AEF]Ⓤ
                    | [#x0B66-#x0B6F]Ⓤ
                    | [#x0BE7-#x0BEF]Ⓤ
                    | [#x0C66-#x0C6F]Ⓤ
                    | [#x0CE6-#x0CEF]Ⓤ
                    | [#x0D66-#x0D6F]Ⓤ
                    | [#x0E50-#x0E59]Ⓤ
                    | [#x0ED0-#x0ED9]Ⓤ
                    | [#x0F20-#x0F29]Ⓤ
[89]       Extender ::= #x00B7Ⓤ | #x02D0
                    | #x02D1Ⓤ | #x0387Ⓤ
                    | #x0640Ⓤ | #x0E46Ⓤ
                    | #x0EC6Ⓤ | #x3005Ⓤ
                    | [#x3031-#x3035]Ⓤ
                    | [#x309D-#x309E]Ⓤ
                    | [#x30FC-#x30FE]Ⓤ
```

The character classes defined here can be derived from the Unicode character database as follows:Ⓤ

- Name start characters must have one of the categories Ll, Lu, Lo, Lt, Nl.Ⓤ
- Name characters other than Name-start characters must have one of the categories Mc, Me, Mn, Lm, or Nd.
- Characters in the compatibility area (i.e. with character code greater than #xF900 and less than #xFFFE) are not allowed in XML names.Ⓣ
- Characters which have a font or compatibility decomposition (i.e. those with a "compatibility formatting tag" in field 5 of the database -- marked by field 5 beginning with a "<") are not allowed.
- The following characters are treated as name-start characters rather than name characters, because the property file classifies them as Alphabetic: [#x02BB-#x02C1], #x0559, #x06E5, #x06E6.
- Characters #x20DD-#x20E0 are excluded (in accordance with Unicode, section 5.14).
- Character #x00B7 is classified as an extender, because the property list so identifies it.
- Character #x0387 is added as a name character, because #x00B7 is its canonical equivalent.
- Characters ':' and '_' are allowed as name-start characters.
- Characters '-' and '.' are allowed as name characters.

C. XML and SGML (Non-Normative)

XML is designed to be a subset of SGML, in that every valid XML document should also be a conformant SGML document. For a detailed comparison of the additional restrictions that XML places on documents beyond those of SGML, see [Clark]. Ⓤ

D. Expansion of Entity and Character References (Non-Normative)

This appendix contains some examples illustrating the sequence of entity- and character-reference recognition and expansion, as specified in "4.4 XML Processor Treatment of Entities and References".

If the DTD contains the declaration

```
<!ENTITY example "<p>An ampersand (&#38;#38;) may be escaped
numerically (&#38;#38;#38;) or with a general entity
(&amp;amp;).</p>" >
```

then the XML processor will recognize the character references when it parses the entity declaration, and resolve them before storing the following string as the value of the entity "example":

```
<p>An ampersand (&#38;) may be escaped
numerically (&#38;#38;) or with a general entity
(&amp;amp;).</p>
```

A reference in the document to "&example;" will cause the text to be reparsed, at which time the start- and end-tags of the "p" element will be recognized and the three references will be recognized and expanded, resulting in a "p" element with the following content (all data, no delimiters or markup):

```
An ampersand (&) may be escaped
numerically (&#38;) or with a general entity
(&amp;).
```

A more complex example will illustrate the rules and their effects fully. In the following example, the line numbers are solely for reference.

```
1 <?xml version='1.0'?>
2 <!DOCTYPE test [
3 <!ELEMENT test (#PCDATA) >
4 <!ENTITY % xx '&#37;zz;'>
5 <!ENTITY % zz '&#60;!ENTITY tricky "error-prone" >' >
6 %xx;
7 ]>
8 <test>This sample shows a &tricky; method.</test>
```

This produces the following:

- in line 4, the reference to character 37 is expanded immediately, and the parameter entity "xx" is stored in the symbol table with the value "%zz;". Since the replacement text is not rescanned, the reference to parameter entity "zz" is not recognized. (And it would be an error if it were, since "zz" is not yet declared.)
- in line 5, the character reference "<" is expanded immediately and the parameter entity "zz" is stored with the replacement text "<!ENTITY tricky "error-prone" >", which is a well-formed entity declaration.
- in line 6, the reference to "xx" is recognized, and the replacement text of "xx" (namely "%zz;") is parsed. The reference to "zz" is recognized in its turn, and its replacement text ("<!ENTITY tricky "error-prone" >") is parsed. The general entity "tricky" has now been declared, with the replacement text "error-prone".
- in line 8, the reference to the general entity "tricky" is recognized, and it is expanded, so the full content of the "test" element is the self-describing (and ungrammatical) string *This sample shows a error-prone method.*

E. Deterministic Content Models (Non-Normative)

For compatibility, it is required that content models in element type declarations be deterministic.

SGML requires deterministic content models (it calls them "unambiguous"); XML processors built using SGML systems may flag non-deterministic content models as errors.

For example, the content model ((b, c) | (b, d)) is non-deterministic, because given an initial b the parser cannot know which b in the model is being matched without looking ahead to see which element follows the b. In this case, the two references to b can be collapsed into a single reference, making the model read (b, (c | d)). An initial b now clearly matches only a single name in the content model. The parser doesn't need to look ahead to see what follows; either c or d would be accepted.

More formally: a finite state automaton may be constructed from the content model using the standard algorithms, e.g. algorithm 3.5 in section 3.9 of Aho, Sethi, and Ullman [Aho/Ullman]. In many such algorithms, a follow set is constructed for each position in the regular expression (i.e., each leaf node in the syntax tree for the regular

expression); if any position has a follow set in which more than one following position is labeled with the same element type name, then the content model is in error and may be reported as an error.

Algorithms exist which allow many but not all non-deterministic content models to be reduced automatically to equivalent deterministic models; see Brüggemann-Klein 1991 [Brüggemann-Klein].

F. Autodetection of Character Encodings (Non-Normative)

The XML encoding declaration functions as an internal label on each entity, indicating which character encoding is in use. Before an XML processor can read the internal label, however, it apparently has to know what character encoding is in use--which is what the internal label is trying to indicate. In the general case, this is a hopeless situation. It is not entirely hopeless in XML, however, because XML limits the general case in two ways: each implementation is assumed to support only a finite set of character encodings, and the XML encoding declaration is restricted in position and content in order to make it feasible to autodetect the character encoding in use in each entity in normal cases. Also, in many cases other sources of information are available in addition to the XML data stream itself. Two cases may be distinguished, depending on whether the XML entity is presented to the processor without, or with, any accompanying (external) information. We consider the first case first.

Because each XML entity not in UTF-8 or UTF-16 format *must* begin with an XML encoding declaration, in which the first characters must be '`<?xml`', any conforming processor can detect, after two to four octets of input, which of the following cases apply. In reading this list, it may help to know that in UCS-4, '<' is "`#x0000003C`" and '?' is "`#x0000003F`", and the Byte Order Mark required of UTF-16 data streams is "`#xFEFF`".

- `00 00 00 3C`: UCS-4, big-endian machine (1234 order)
- `3C 00 00 00`: UCS-4, little-endian machine (4321 order)
- `00 00 3C 00`: UCS-4, unusual octet order (2143)
- `00 3C 00 00`: UCS-4, unusual octet order (3412)
- `FE FF`: UTF-16, big-endian
- `FF FE`: UTF-16, little-endian
- `00 3C 00 3F`: UTF-16, big-endian, no Byte Order Mark (and thus, strictly speaking, in error)
- `3C 00 3F 00`: UTF-16, little-endian, no Byte Order Mark (and thus, strictly speaking, in error)
- `3C 3F 78 6D`: UTF-8, ISO 646, ASCII, some part of ISO 8859, Shift-JIS, EUC, or any other 7-bit, 8-bit, or mixed-width encoding which ensures that the characters of ASCII have their normal positions, width, and values; the actual encoding declaration must be read to detect which of these applies, but since all of these encodings use the same bit patterns for the ASCII characters, the encoding declaration itself may be read reliably
- `4C 6F A7 94`: EBCDIC (in some flavor; the full encoding declaration must be read to tell which code page is in use)
- other: UTF-8 without an encoding declaration, or else the data stream is corrupt, fragmentary, or enclosed in a wrapper of some kind

This level of autodetection is enough to read the XML encoding declaration and parse the character-encoding identifier, which is still necessary to distinguish the individual members of each family of encodings (e.g. to tell UTF-8 from 8859, and the parts of 8859 from each other, or to distinguish the specific EBCDIC code page in use, and so on).

Because the contents of the encoding declaration are restricted to ASCII characters, a processor can reliably read the entire encoding declaration as soon as it has detected which family of encodings is in use. Since in practice, all widely used character encodings fall into one of the categories above, the XML encoding declaration allows reasonably reliable in-band labeling of character encodings, even when external sources of information at the operating-system or transport-protocol level are unreliable.

Once the processor has detected the character encoding in use, it can act appropriately, whether by invoking a separate input routine for each case, or by calling the proper conversion function on each character of input.

Like any self-labeling system, the XML encoding declaration will not work if any software changes the entity's character set or encoding without updating the encoding declaration. Implementors of character-encoding routines should be careful to ensure the accuracy of the internal and external information used to label the entity.

The second possible case occurs when the XML entity is accompanied by encoding information, as in some file systems and some network protocols. When multiple sources of information are available, their relative priority and the preferred method of handling conflict should be specified as part of the higher-level protocol used to deliver XML. Rules for the relative priority of the internal label and the MIME-type label in an external header, for example, should be part of the RFC document defining the text/xml and application/xml MIME types. In the interests of interoperability, however, the following rules are recommended.

- If an XML entity is in a file, the Byte-Order Mark and encoding-declaration PI are used (if present) to determine the character encoding. All other heuristics and sources of information are solely for error recovery.
- If an XML entity is delivered with a MIME type of text/xml, then the `charset` parameter on the MIME type determines the character encoding method; all other heuristics and sources of information are solely for error recovery.
- If an XML entity is delivered with a MIME type of application/xml, then the Byte-Order Mark and

encoding-declaration PI are used (if present) to determine the character encoding. All other heuristics and sources of information are solely for error recovery.

These rules apply only in the absence of protocol-level documentation; in particular, when the MIME types text/xml and application/xml are defined, the recommendations of the relevant RFC will supersede these rules.

G. W3C XML Working Group (Non-Normative)

This specification was prepared and approved for publication by the W3C XML Working Group (WG). WG approval of this specification does not necessarily imply that all WG members voted for its approval. The current and former members of the XML WG are:

Jon Bosak, Sun (Chair); James Clark (Technical Lead); Tim Bray, Textuality and Netscape (XML Co-editor); Jean Paoli, Microsoft (XML Co-editor); C. M. Sperberg-McQueen, U. of Ill. (XML Co-editor); Dan Connolly, W3C (W3C Liaison); Paula Angerstein, Texcel; Steve DeRose, INSO; Dave Hollander, HP; Eliot Kimber, ISOGEN; Eve Maler, ArborText; Tom Magliery, NCSA; Murray Maloney, Muzmo and Grif; Makoto Murata, Fuji Xerox Information Systems; Joel Nava, Adobe; Conleth O'Connell, Vignette; Peter Sharpe, SoftQuad; John Tigue, DataChannel

O'REILLY xml.com
xml from the inside out

LINUX
Streaming Audio With Linux

Find it on O'Reilly Network

Designing JSP Custom Tag Libraries

Find it at ONJava.com

A Technical Introduction to XML

Print This Article

by Norman Walsh
October 03, 1998

This introduction to XML presents the Extensible Markup Language at a reasonably technical level for anyone interested in learning more about structured documents. In addition to covering the XML 1.0 Specification, this article outlines related XML specifications, which are evolving. The article is organized in four main sections plus an appendix.

Start Here

Introduction
What is XML?
What's a Document?
So XML is Just Like HTML?
So XML Is Just Like SGML?

Author's Note

It is somewhat remarkable to think that this article, which appeared initially in the Winter 1997 edition of the World Wide Web Journal was out of date by the time the final XML Recommendation was approved in February. And even as this update brings the article back into line with the final spec, a new series of recommendations are under development. When finished, these will bring namespaces, linking, schemas, stylesheets, and more to the table.

O'REILLY OPEN SOURCE CONVENTION
July 23–27, 2001
San Diego

conferences.oreilly.com
INNOVATE
COLLABORATE
DISCOVER

Sponsored By:

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

Why XML?
XML Development Goals
How Is XML Defined?
Understanding the Specs

What Do XML Documents Look Like?
Elements
Entity References
Comments
Processing Instructions
CDATA Sections
Document Type Declarations
Other Markup Issues

Validity
Well-formed Documents
Valid Documents

Pulling the Pieces Together
Simple Links
Extended Links
Extended Pointers
Extended Link Groups
Understanding The Pieces
Style and Substance
Conclusion

Appendix:
Extended Backus-Naur Form (EBNF)
Revision History

Sponsored By:

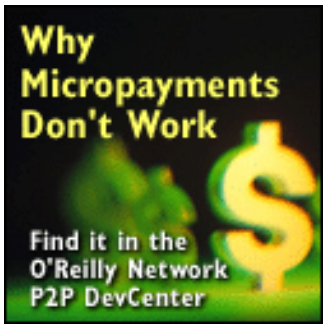
Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Ximian Gnome

Find it on the O'Reilly Network
Welcome to Your Desktop!



What is XML?

Print This Article

by Norman Walsh
October 03, 1998

XML is a markup language for documents containing structured information.

Structured information contains both content (words, pictures, etc.) and some indication of what role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.). Almost all documents have some structure.

A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents.

What's a Document?

The number of applications currently being developed that are based on, or make use of, XML documents is truly amazing (particularly when you consider that XML is not yet a year old)! For our purposes, the word "document" refers not only to traditional documents, like this one, but also to the miriad of other XML "data formats". These include vector graphics, e-commerce transactions, mathematical equations, object meta-data, server APIs, and a thousand other kinds of structured information.

So XML is Just Like HTML?

No. In HTML, both the tag semantics and the tag set are fixed. An `<h1>` is always a first

Sponsored By:

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES
XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX
Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

level heading and the tag `<ati.product.code>` is meaningless. The W3C, in conjunction with browser vendors and the WWW community, is constantly working to extend the definition of HTML to allow new tags to keep pace with changing technology and to bring variations in presentation (stylesheets) to the Web. However, these changes are always rigidly confined by what the browser vendors have implemented and by the fact that backward compatibility is paramount. And for people who want to disseminate information widely, features supported by only the latest releases of Netscape and Internet Explorer are not useful.

XML specifies neither semantics nor a tag set. In fact XML is really a meta-language for describing markup languages. In other words, XML provides a facility to define tags and the structural relationships between them. Since there's no predefined tag set, there can't be any preconceived semantics. All of the semantics of an XML document will either be defined by the applications that process them or by stylesheets.

So XML Is Just Like SGML?

No. Well, yes, sort of. XML is defined as an application profile of SGML. SGML is the Standard Generalized Markup Language defined by ISO 8879. SGML has been the standard, vendor-independent way to maintain repositories of structured documentation for more than a decade, but it is not well suited to serving documents over the web (for a number of technical reasons beyond the scope of this article). Defining XML as an application profile of SGML means that any fully conformant SGML system will be able to read XML documents. However, using and understanding XML documents *does not* require a system that is capable of understanding the full generality of SGML. XML is, roughly speaking, a restricted form of SGML.

For technical purists, it's important to note that there may also be subtle differences between documents as understood by XML systems and those same documents as understood by SGML systems. In particular, treatment of white space immediately adjacent to tags may be different.

Sponsored By:

Why XML?

In order to appreciate XML, it is important to understand why it was created. XML was created so that richly structured documents could be used over the web. The only viable alternatives, HTML and SGML, are not practical for this purpose.

HTML, as we've already discussed, comes bound with a set of semantics and does not provide arbitrary structure.

SGML provides arbitrary structure, but is too difficult to implement just for a web browser. Full SGML systems solve large, complex problems that justify their expense. Viewing structured documents sent over the web rarely carries such justification.

This is not to say that XML can be expected to completely replace SGML. While XML is being designed to deliver structured content over the web, some of the very features it lacks to make this practical, make SGML a more satisfactory solution for the creation and long-time storage of complex documents. In many organizations, filtering SGML to XML will be the standard procedure for web delivery.

XML Development Goals

The XML specification sets out the following goals for XML: [Section 1.1] (In this article, citations of the form [Section 1.1], these are references to the W3C Recommendation Extensible Markup Language (XML) 1.0. If you are interested in more technical detail about a particular topic, please consult the specification)

1. It shall be straightforward to use XML over the Internet. Users must be able to view XML documents as quickly and easily as HTML documents. In practice, this will only be possible when XML browsers are as robust and widely available as HTML browsers, but the principle remains.
2. XML shall support a wide variety of applications. XML should be beneficial to a wide variety of diverse applications:

authoring, browsing, content analysis, etc. Although the initial focus is on serving structured documents over the web, it is not meant to narrowly define XML.

3. XML shall be compatible with SGML. Most of the people involved in the XML effort come from organizations that have a large, in some cases staggering, amount of material in SGML. XML was designed pragmatically, to be compatible with existing standards while solving the relatively new problem of sending richly structured documents over the web.
4. It shall be easy to write programs that process XML documents. The colloquial way of expressing this goal while the spec was being developed was that it ought to take about two weeks for a competent computer science graduate student to build a program that can process XML documents.
5. The number of optional features in XML is to be kept to an absolute minimum, ideally zero. Optional features inevitably raise compatibility problems when users want to share documents and sometimes lead to confusion and frustration.
6. XML documents should be human-legible and reasonably clear. If you don't have an XML browser and you've received a hunk of XML from somewhere, you ought to be able to look at it in your favorite text editor and actually figure out what the content means.
7. The XML design should be prepared quickly. Standards efforts are notoriously slow. XML was needed immediately and was developed as quickly as possible.
8. The design of XML shall be formal and concise. In many ways a corollary to rule 4, it essentially means that XML must be expressed in EBNF and must be amenable to modern compiler tools and techniques.
   There are a number of technical reasons why the SGML grammar *cannot* be expressed in EBNF. Writing a proper

SGML parser requires handling a variety of rarely used and difficult to parse language features. XML does not.

9. XML documents shall be easy to create. Although there will eventually be sophisticated editors to create and edit XML content, they won't appear immediately. In the interim, it must be possible to create XML documents in other ways: directly in a text editor, with simple shell and Perl scripts, etc.
10. Terseness in XML markup is of minimal importance. Several SGML language features were designed to minimize the amount of typing required to manually key in SGML documents. These features are not supported in XML. From an abstract point of view, these documents are indistinguishable from their more fully specified forms, but supporting these features adds a considerable burden to the SGML parser (or the person writing it, anyway). In addition, most modern editors offer better facilities to define shortcuts when entering text.

How Is XML Defined?

XML is defined by a number of related specifications:

Extensible Markup Language (XML) 1.0

Defines the syntax of XML. The XML specification is the primary focus of this article.

XML Pointer Language (XPointer) and XML Linking Language (XLink)

Defines a standard way to represent links between resources. In addition to simple links, like HTML's <A> tag, XML has mechanisms for links between multiple resources and links between read-only resources. XPointer describes how to address a resource, XLink describes how to associate two or more resources.

Extensible Style Language (XSL)

Defines the standard stylesheet language for XML.

As time goes on, additional requirements will

be addressed by other specifications. Currently (Sep, 1998), namespaces (dealing with tags from multiple tag sets), a query language (finding out what's in a document or a collection of documents), and a schema language (describing the relationships between tags, DTDs in XML) are all being actively pursued.

Understanding the Specs

For the most part, reading and understanding the XML specifications does not require extensive knowledge of SGML or any of the related technologies.

One topic that may be new is the use of EBNF to describe the syntax of XML. Please consult the discussion of EBNF in the appendix of this article for a detailed description of how this grammar works.

Next: What Do XML Documents Look Like?

Copyright © 2001 O'Reilly & Associates, Inc.

O'REILLY xml.com
xml from the inside out

Resource Description Format
RDF Applications with Prolog
Find it on XML.com

XML.COM | O'REILLY NETWORK | OREILLY.COM | ONJAVA.COM

Home | Resources | Buyer's Guide | FAQs | Free Newsletter
search

RESOURCE CENTERS
Business
Graphics
Metadata
Mobile
Programming
Protocols
Schemas
Style
Web

ESSENTIALS
Annotated XML
What is XML?
What is XSLT?
What is XLink?
What is XML Schema?
What is RDF?

SSH Tunneling Secures 802.11
In the Wireless DevCenter

FIND
Search
Article Archive
FAQs

Add the latest news from XML.com to your web site
Click here!

COLUMNS
XML-Deviant
Style Matters
XML Q&A
Transforming XML
Perl and XML

GUIDES

What Do XML Documents Look Like?

Print This Article

by Norman Walsh
October 03, 1998

If you are conversant with HTML or SGML, XML documents will look familiar. A simple XML document is presented in Example 1.

Example 1. A Simple XML Document

```
<?xml version="1.0"?>

<oldjoke>

<burns>Say <quote>goodnight</quote>,
Gracie.</burns>

<allen><quote>Goodnight,
Gracie.</quote></allen>

<applause/>

</oldjoke>
```

A few things may stand out to you:

- The document begins with a processing instruction: `<?xml ...?>`. This is the *XML declaration* [Section 2.8]. While it is not required, its presence explicitly identifies the document as an XML document and indicates the version of XML to which it was authored.
- There's no document type declaration. Unlike SGML, XML does not require a document type declaration. However, a document type declaration can be supplied, and some documents will require one in order to be understood unambiguously.
- Empty elements (`<applause/>` in this example) have a modified syntax [Section 3.1]. While most elements in a document are wrappers around some content, empty elements are simply markers where something occurs (a horizontal rule for HTML's `<hr>` tag, for example, or a cross reference for DocBook's `<xref>` tag). The trailing /> in the modified syntax indicates to a program processing the XML document that the element is empty and no matching end-tag should be sought. Since XML documents do not require a document type declaration, without this clue it could be impossible for an XML parser to determine which tags were intentionally empty and which had been left empty by mistake.
  XML has softened the distinction between elements which are declared as `EMPTY` and elements which merely have no content. In XML, it is legal to use the empty-element tag syntax in either case. It's also legal to use a start-tag/end-tag pair for empty elements: `<applause></applause>`. If interoperability is of any concern, it's best to reserve empty-element tag syntax for elements which are

XML
Products
Tools
Training
O'REILLY xml.com

XML Resources
Buyer's Guide
Events Calendar
Standards List
Submissions List

TOOLBOX

Syntax Checker
XML Testbed

SEYBOLD PUBLICATIONS

THIS SITE IS ON THE O'REILLY NETWORK

declared as `EMPTY` and to only use the empty-element tag form for those elements.

XML documents are composed of markup and content. There are six kinds of markup that can occur in an XML document: elements, entity references, comments, processing instructions, marked sections, and document type declarations. The following sections introduce each of these markup concepts.

Elements

Elements are the most common form of markup. Delimited by angle brackets, most elements identify the nature of the content they surround. Some elements may be empty, as seen above, in which case they have no content. If an element is not empty, it begins with a start-tag, `<element>`, and ends with an end-tag, `</element>`.

Attributes

Attributes are name-value pairs that occur inside start-tags after the element name. For example,

```
<div class="preface">
```

is a `div` element with the attribute `class` having the value `preface`. In XML, all attribute values must be quoted.

Entity References

In order to introduce markup into a document, some characters have been reserved to identify the start of markup. The left angle bracket, < , for instance, identifies the beginning of an element start- or end-tag. In order to insert these characters into your document as content, there must be an alternative way to represent them. In XML, entities are used to represent these special characters. Entities are also used to refer to often repeated or varying text and to include the content of external files.

Every entity must have a unique name. Defining your own entity names is discussed in the section on entity declarations. In order to use an entity, you simply reference it by name. Entity references begin with the ampersand and end with a semicolon.

For example, the `lt` entity inserts a literal < into a document. So the string `<element>` can be represented in an XML document as `<element>`.

A special form of entity reference, called a character reference [Section 4.1], can be used to insert arbitrary Unicode characters into your document. This is a mechanism for inserting characters that cannot be typed directly on your keyboard.

Character references take one of two forms: decimal references, `℞`, and hexadecimal references, `℞`. Both of these refer to character number U+211E from Unicode (which is the standard Rx prescription symbol, in case you were wondering).

Comments

Comments begin with `<!--` and end with `-->`. Comments can contain any data except the literal string `--`. You can place comments between markup anywhere in your document.

Comments are not part of the textual content of an XML document. An XML processor is not required to pass them along to an application.

Processing Instructions

Processing instructions (PIs) are an escape hatch to provide information to an

application. Like comments, they are not textually part of the XML document, but the XML processor is required to pass them to an application.

Processing instructions have the form: `<?name pidata?>`. The name, called the PI target, identifies the PI to the application. Applications should process only the targets they recognize and ignore all other PIs. Any data that follows the PI target is optional, it is for the application that recognizes the target. The names used in PIs may be declared as notations in order to formally identify them.

PI names beginning with `xml` are reserved for XML standardization.

CDATA Sections

In a document, a `CDATA` section instructs the parser to ignore most markup characters.

Consider a source code listing in an XML document. It might contain characters that the XML parser would ordinarily recognize as markup (< and &, for example). In order to prevent this, a CDATA section can be used.

```
<![CDATA[

*p = &q;

b = (i <= 3);

]]>
```

Between the start of the section, `<![CDATA[` and the end of the section, `]]>`, all character data is passed directly to the application, without interpretation. Elements, entity references, comments, and processing instructions are all unrecognized and the characters that comprise them are passed literally to the application.

The only string that cannot occur in a CDATA section is `]]>`.

Document Type Declarations

A large percentage of the XML specification deals with various sorts of declarations that are allowed in XML. If you have experience with SGML, you will recognize these declarations from SGML DTDs (Document Type Definitions). If you have never seen them before, their significance may not be immediately obvious.

One of the greatest strengths of XML is that it allows you to create your own tag names. But for any given application, it is probably not meaningful for tags to occur in a completely arbitrary order. Consider the old joke example introduced earlier. Would this be meaningful?

```
<gracie><quote><oldjoke>Goodnight,
<applause/>Gracie</oldjoke></quote>

<burns><gracie>Say <quote>goodnight</quote>,
</gracie>Gracie.</burns></gracie>
```

It's so far outside the bounds of what we normally expect that it's nonsensical. It just doesn't *mean* anything.

However, from a strictly syntactic point of view, there's nothing wrong with that XML document. So, if the document is to have meaning, and certainly if you're writing a stylesheet or application to process it, there must be some constraint on the sequence and nesting of tags. Declarations are where these constraints can be expressed.

More generally, declarations allow a document to communicate

meta-information to the parser about its content. Meta-information includes the allowed sequence and nesting of tags, attribute values and their types and defaults, the names of external files that may be referenced and whether or not they contain XML, the formats of some external (non-XML) data that may be referenced, and the entities that may be encountered.

There are four kinds of declarations in XML: element type declarations, attribute list declarations, entity declarations, and notation declarations.

Element Type Declarations

Element type declarations [Section 3.2] identify the names of elements and the nature of their content. A typical element type declaration looks like this:

```
<!ELEMENT oldjoke (burns+, allen, applause?)>
```

This declaration identifies the element named `oldjoke`. Its *content model* follows the element name. The content model defines what an element may contain. In this case, an `oldjoke` must contain `burns` and `allen` and may contain `applause`. The commas between element names indicate that they must occur in succession. The plus after `burns` indicates that it may be repeated more than once but must occur at least once. The question mark after `applause` indicates that it is optional (it may be absent, or it may occur exactly once). A name with no punctuation, such as `allen`, must occur exactly once.

Declarations for `burns`, `allen`, `applause` and all other elements used in any content model must also be present for an XML processor to check the validity of a document.

In addition to element names, the special symbol `#PCDATA` is reserved to indicate character data. The moniker PCDATA stands for parseable character data .

Elements that contain only other elements are said to have *element content* [Section 3.2.1]. Elements that contain both other elements and `#PCDATA` are said to have *mixed content* [Section 3.2.2].

For example, the definition for `burns` might be

```
<!ELEMENT burns (#PCDATA | quote)*>
```

The vertical bar indicates an or relationship, the asterisk indicates that the content is optional (may occur zero or more times); therefore, by this definition, `burns` may contain zero or more characters and `quote` tags, mixed in any order. All mixed content models must have this form: `#PCDATA` must come first, all of the elements must be separated by vertical bars, and the entire group must be optional.

Two other content models are possible: `EMPTY` indicates that the element has no content (and consequently no end-tag), and `ANY` indicates that *any* content is allowed. The `ANY` content model is sometimes useful during document conversion, but should be avoided at almost any cost in a production environment because it disables all content checking in that element.

Here is a complete set of element declarations for Example 1:

Example 2. Element Declarations for Old Jokes

```
<!ELEMENT oldjoke  (burns+, allen, applause?)>

<!ELEMENT burns    (#PCDATA | quote)*>

<!ELEMENT allen    (#PCDATA | quote)*>

<!ELEMENT quote    (#PCDATA)*>

<!ELEMENT applause EMPTY>
```

Attribute List Declarations

Attribute list declarations [Section 3.3] identify which elements may have attributes, what attributes they may have, what values the attributes may hold, and what value is the default. A typical attribute list declaration looks like this:

```
<!ATTLIST oldjoke

    name
ID
#REQUIRED

    label
CDATA
#IMPLIED

    status ( funny | notfunny ) 'funny'>
```

In this example, the `oldjoke` element has three attributes: `name`, which is an ID and is required; `label`, which is a string (character data) and is not required; and `status`, which must be either `funny` or `notfunny` and defaults to `funny`, if no value is specified.

Each attribute in a declaration has three parts: a name, a type, and a default value.

You are free to select any name you wish, subject to some slight restrictions [Section 2.3, production 5], but names cannot be repeated on the same element.

There are six possible attribute types:

`CDATA`

> CDATA attributes are strings, any text is allowed. Don't confuse CDATA attributes with CDATA sections, they are unrelated.

`ID`

> The value of an ID attribute must be a name [Section 2.3, production 5]. All of the ID values used in a document must be different. IDs uniquely identify individual elements in a document. Elements can have only a single ID attribute.

`IDREF`

or `IDREFS`

> An IDREF attribute's value must be the value of a single ID attribute on some element in the document. The value of an IDREFS attribute may contain multiple IDREF values separated by white space [Section 2.3, production 3].

`ENTITY`

or `ENTITIES`

> An ENTITY attribute's value must be the name of a single entity (see the discussion of entity declarations below). The value of an ENTITIES attribute may contain multiple entity names separated by white space.

`NMTOKEN`

or `NMTOKENS`

> Name token attributes are a restricted form of string attribute. In general, an NMTOKEN attribute must consist of a single word [Section 2.3, production 7], but there are no additional constraints on the word, it doesn't have to match another attribute or declaration. The

value of an NMTOKENS attribute may contain multiple NMTOKEN values separated by white space.

A list of names

You can specify that the value of an attribute must be taken from a specific list of names. This is frequently called an enumerated type because each of the possible values is explicitly enumerated in the declaration.

Alternatively, you can specify that the names must match a notation name (see the discussion of notation declarations below).

There are four possible default values:

`#REQUIRED`

The attribute must have an explicitly specified value on every occurrence of the element in the document.

`#IMPLIED`

The attribute value is not required, and no default value is provided. If a value is not specified, the XML processor must proceed without one.

"value"

An attribute can be given any legal value as a default. The attribute value is not required on each element in the document, and if it is not present, it will appear to be the specified default.

`#FIXED`

"value"

An attribute declaration may specify that an attribute has a fixed value. In this case, the attribute is not required, but if it occurs, it must have the specified value. If it is not present, it will appear to be the specified default. One use for fixed attributes is to associate semantics with an element. A complete discussion is beyond the scope of this article, but you can find several examples of fixed attributes in the XLink specification.

The XML processer performs *attribute value normalization* [Section 3.3.3] on attribute values: character references are replaced by the referenced character, entity references are resolved (recursively), and whitespace is normalized.

Entity Declarations

Entity declarations [Section 4.2] allow you to associate a name with some other fragment of content. That construct can be a chunk of regular text, a chunk of the document type declaration, or a reference to an external file containing either text or binary data.

A few typical entity declarations are shown in Example 3.

Example 3. Typical Entity Declarations

```
<!ENTITY
ATI
"ArborText, Inc.">

<!ENTITY boilerplate     SYSTEM
"/standard/legalnotice.xml">

<!ENTITY ATIlogo
SYSTEM "/standard/logo.gif" NDATA GIF87A>
```

There are three kinds of entities:

Internal Entities

Internal entities [Section 4.2.1] associate a name with a string of literal

text. The first entity in Example 3 is an internal entity. Using `&ATI;` anywhere in the document will insert ArborText, Inc. at that location. Internal entities allow you to define shortcuts for frequently typed text or text that is expected to change, such as the revision status of a document.

Internal entities can include references to other internal entities, but it is an error for them to be recursive.

The XML specification predefines five internal entities:

- `<` produces the left angle bracket, <
- `>` produces the right angle bracket, >
- `&` produces the ampersand, &
- `'` produces a single quote character (an apostrophe), '
- `"` produces a double quote character, "

External Entities

External entities [Section 4.2.2] associate a name with the content of another file. External entities allow an XML document to refer to the contents of another file. External entities contain either text or binary data. If they contain text, the content of the external file is inserted at the point of reference and parsed as part of the referring document. Binary data is not parsed and may only be referenced in an attribute. Binary data is used to reference figures and other non-XML content in the document.

The second and third entities in Example 3 are external entities.

Using `&boilerplate;` will have insert the *contents* of the file `/standard/legalnotice.xml` at the location of the entity reference. The XML processor will parse the content of that file as if it occurred literally at that location.

The entity `ATIlogo` is also an external entity, but its content is binary. The `ATIlogo` entity can only be used as the value of an ENTITY (or ENTITIES) attribute (on a `graphic` element, perhaps). The XML processor will pass this information along to an application, but it does not attempt to process the content of `/standard/logo.gif`.

Parameter Entities

Parameter entities can only occur in the document type declaration. A parameter entity declaration is identified by placing `%` (percent-space) in front of its name in the declaration. The percent sign is also used in references to parameter entities, instead of the ampersand. Parameter entity references are immediately expanded in the document type declaration and their replacement text is part of the declaration, whereas normal entity references are not expanded. Parameter entities are not recognized in the body of a document.

Looking back at the element declarations in Example 2, you'll notice that two of them have the same content model:

```
<!ELEMENT burns    (#PCDATA | quote)*>

<!ELEMENT allen    (#PCDATA | quote)*>
```

At the moment, these two elements are the same only because they happen to have the same literal definition. In order to make more explicit the fact that these two elements are semantically the same, use a parameter entity to define their content model. The advantage of using a parameter entity is two-fold. First, it allows you to give a descriptive name to the content, and second it allows you to change the content model in only a single place, if you wish to update the element declarations, assuring that they always stay the same:

```
<!ENTITY % personcontent "#PCDATA | quote">
```

```
<!ELEMENT burns (%personcontent;)*>

<!ELEMENT allen (%personcontent;)*>
```

Notation Declarations

Notation declarations [Section 4.7] identify specific types of external binary data. This information is passed to the processing application, which may make whatever use of it it wishes. A typical notation declaration is:

```
<!NOTATION GIF87A SYSTEM "GIF">
```

Do I need a Document Type Declaration?

As we've seen, XML content can be processed without a document type declaration. However, there are some instances where the declaration is required:

Authoring Environments

> Most authoring environments need to read and process document type declarations in order to understand and enforce the content models of the document.

Default Attribute Values

> If an XML document relies on default attribute values, at least part of the declaration must be processed in order to obtain the correct default values.

White Space Handling

> The semantics associated with white space in element content differs from the semantics associated with white space in mixed content. Without a DTD, there is no way for the processor to distinguish between these cases, and all elements are effectively mixed content. For more detail, see the section called *White Space Handling*, later in this document.

In applications where a person composes or edits the data (as opposed to data that may be generated directly from a database, for example), a DTD is probably going to be required if any structure is to be guaranteed.

Including a Document Type Declaration

If present, the document type declaration must be the first thing in the document after optional processing instructions and comments [Section 2.8].

The document type declaration identifies the root element of the document and may contain additional declarations. All XML documents must have a single root element that contains all of the content of the document. Additional declarations may come from an external DTD, called the external subset, or be included directly in the document, the internal subset, or both:

```
<?XML version="1.0" standalone="no"?>

<!DOCTYPE chapter SYSTEM "dbook.dtd" [

<!ENTITY %ulink.module "IGNORE">

<!ELEMENT ulink (#PCDATA)*>

<!ATTLIST ulink

    xml:link       CDATA  #FIXED "SIMPLE"

    xml-attributes CDATA  #FIXED "HREF URL"

    URL            CDATA  #REQUIRED>
```

```
]>

<chapter>...</chapter>
```

This example references an external DTD, `dbook.dtd`, and includes element and attribute declarations for the `ulink` element in the internal subset. In this case, `ulink` is being given the semantics of a simple link from the XLink specification.

Note that declarations in the internal subset override declarations in the external subset. The XML processor reads the internal subset before the external subset and the *first* declaration takes precedence.

In order to determine if a document is valid, the XML processor must read the entire document type declaration (both internal and external subsets). But for some applications, validity may not be required, and it may be sufficient for the processor to read only the internal subset. In the example above, if validity is unimportant and the only reason to read the doctype declaration is to identify the semantics of `ulink`, reading the external subset is not necessary.

You can communicate this information in the *standalone document declaration* [Section 2.9]. The standalone document declaration, `standalone="yes"` or `standalone="no"` occurs in the XML declaration. A value of `yes` indicates that only internal declarations need to be processed. A value of `no` indicates that *both* the internal and external declarations must be processed.

Other Markup Issues

In addition to markup, there are a few other issues to consider: white space handling, attribute value normalization, and the language in which the document is written.

White Space Handling

White space handling [Section 2.10] is a subtle issue. Consider the following content fragment:

```
<oldjoke>

<burns>Say <quote>goodnight</quote>, Gracie.</burns>
```

Is the white space (the new line between `<oldjoke>` and `<burns>`) significant?

Probably not.

But how can you tell? You can only determine if white space is significant if you know the content model of the elements in question. In a nutshell, white space is significant in mixed content and is insignificant in element content.

The rule for XML processors is that they must pass all characters that are not markup through to the application. If the processor is a validating processor [Section 5.1], it must also inform the application about which whitespace characters are significant.

The special attribute `xml:space` may be used to indicate explicitly that white space is significant. On any element which includes the attribute specification `xml:space='preserve'`, all white space within that element (and within subelements that do not explicitly reset `xml:space`) is significant.

The only legal values for `xml:space` are `preserve` and `default`. The value `default` indicates that the default processing is desired. In a DTD, the `xml:space` attribute must be declared as an enumerated type with only

those two values.

One last note about white space: in parsed text, XML processors are required to normalize all end-of-line markers to a single line feed character (&#A;) [Section 2.11]. This is rarely of interest to document authors, but it does eliminate a number of cross-platform portability issues.

Attribute Value Normalization

The XML processer performs attribute value normalization [Section 3.3.3] on attribute values: character references are replaced by the referenced character, entity references are resolved (recursively), and whitespace is normalized.

Language Identification

Many document processing applications can benefit from information about the natural language in which a document is written, XML defines the attribute `xml:lang` [Section 2.12] to identify the language. Since the purpose of this attribute is to standardize information across applications, the XML specification also describes how languages are to be identified.

Previous: What is XML? ▶ **Next: Validity** ▶

Contact Us | Our Mission | Privacy Policy | Advertise With Us | Site Help

Copyright © 2001 O'Reilly & Associates, Inc.

Extensible Markup Language (XML) 1.0 (Second Edition)

W3C Recommendation 6 October 2000

This version:

http://www.w3.org/TR/2000/REC-xml-20001006 (XHTML, XML, PDF, XHTML review version with color-coded revision indicators)

Latest version:

http://www.w3.org/TR/REC-xml

Previous versions:

http://www.w3.org/TR/2000/WD-xml-2e-20000814

http://www.w3.org/TR/1998/REC-xml-19980210

Editors:

Tim Bray, Textuality and Netscape <tbray@textuality.com>

Jean Paoli, Microsoft <jeanpa@microsoft.com>

C. M. Sperberg-McQueen, University of Illinois at Chicago and Text Encoding Initiative <cmsmcq@uic.edu>

Eve Maler, Sun Microsystems, Inc. <eve.maler@east.sun.com> - Second Edition

Copyright © 2000 W3C® (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use, and software licensing rules apply.

Abstract

The Extensible Markup Language (XML) is a subset of SGML that is completely described in this document. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

Status of this Document

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document specifies a syntax created by subsetting an existing, widely used international text processing standard (Standard Generalized Markup Language, ISO 8879:1986(E) as amended and corrected) for use on the World Wide Web. It is a product of the W3C XML Activity, details of which can be found at http://www.w3.org/XML. The English version of this specification is the only normative version. However, for translations of this document, see http://www.w3.org/XML/#trans. A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

This second edition is *not* a new version of XML (first published 10 February 1998); it merely incorporates the changes dictated by the first-edition errata (available at http://www.w3.org/XML/xml-19980210-errata) as a convenience to readers. The errata list for this second edition is available at http://www.w3.org/XML/xml-V10-2e-errata.

Please report errors in this document to xml-editor@w3.org; archives are available.

Note:

C. M. Sperberg-McQueen's affiliation has changed since the publication of the first edition. He is now at the World Wide Web Consortium, and can be contacted at cmsmcq@w3.org.

Table of Contents

1 Introduction
  1.1 Origin and Goals
  1.2 Terminology
2 Documents
  2.1 Well-Formed XML Documents
  2.2 Characters
  2.3 Common Syntactic Constructs
  2.4 Character Data and Markup
  2.5 Comments
  2.6 Processing Instructions
  2.7 CDATA Sections
  2.8 Prolog and Document Type Declaration
  2.9 Standalone Document Declaration
  2.10 White Space Handling
  2.11 End-of-Line Handling
  2.12 Language Identification
3 Logical Structures
  3.1 Start-Tags, End-Tags, and Empty-Element Tags
  3.2 Element Type Declarations
    3.2.1 Element Content
    3.2.2 Mixed Content
  3.3 Attribute-List Declarations
    3.3.1 Attribute Types
    3.3.2 Attribute Defaults
    3.3.3 Attribute-Value Normalization
  3.4 Conditional Sections
4 Physical Structures
  4.1 Character and Entity References
  4.2 Entity Declarations
    4.2.1 Internal Entities
    4.2.2 External Entities
  4.3 Parsed Entities
    4.3.1 The Text Declaration
    4.3.2 Well-Formed Parsed Entities
    4.3.3 Character Encoding in Entities
  4.4 XML Processor Treatment of Entities and References
    4.4.1 Not Recognized

4.4.2 Included
4.4.3 Included If Validating
4.4.4 Forbidden
4.4.5 Included in Literal
4.4.6 Notify
4.4.7 Bypassed
4.4.8 Included as PE
4.5 Construction of Internal Entity Replacement Text
4.6 Predefined Entities
4.7 Notation Declarations
4.8 Document Entity
5 Conformance
5.1 Validating and Non-Validating Processors
5.2 Using XML Processors
6 Notation

Appendices

A References
A.1 Normative References
A.2 Other References
B Character Classes
C XML and SGML (Non-Normative)
D Expansion of Entity and Character References (Non-Normative)
E Deterministic Content Models (Non-Normative)
F Autodetection of Character Encodings (Non-Normative)
F.1 Detection Without External Encoding Information
F.2 Priorities in the Presence of External Encoding Information
G W3C XML Working Group (Non-Normative)
H W3C XML Core Group (Non-Normative)
I Production Notes (Non-Normative)

---

1 Introduction

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]. By construction, XML documents are conforming SGML documents.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

[Definition: A software module called an **XML processor** is used to read XML documents and provide access to their content and structure.] [Definition: It is assumed that an XML processor is doing its work on behalf of another module, called the **application**.] This specification describes the required behavior of an XML processor in terms of how it must read XML data and the information it must provide to the application.

1.1 Origin and Goals

XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996. It was chaired by Jon Bosak of Sun Microsystems with the active participation of an XML Special Interest Group (previously known as the SGML Working Group) also organized by the W3C. The membership of the XML Working Group is given in an appendix. Dan Connolly served as the WG's contact with the W3C.

The design goals for XML are:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

This specification, together with associated standards (Unicode and ISO/IEC 10646 for characters, Internet RFC 1766 for language identification tags, ISO 639 for language name codes, and ISO 3166 for country name codes), provides all the information necessary to understand XML Version 1.0 and construct computer programs to process it.

This version of the XML specification may be distributed freely, as long as all text and legal notices remain intact.

1.2 Terminology

The terminology used to describe XML documents is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of an XML processor:

may

> [Definition: Conforming documents and XML processors are permitted to but need not behave as described.]

must

> [Definition: Conforming documents and XML processors are required to behave as described; otherwise they are in error.]

error

> [Definition: A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it.]

fatal error

> [Definition: An error which a conforming XML processor must detect and report to the application. After encountering a fatal error, the processor may continue processing the data to search for further errors and may report such errors to the application. In order to support correction of errors, the processor may make unprocessed data from the document (with intermingled character data and markup) available to the application. Once a fatal error is detected, however, the processor must not continue normal processing (i.e., it must not continue to pass character data and information about the document's logical structure to the application in the normal way).]

at user option

[Definition: Conforming software may or must (depending on the modal verb in the sentence) behave as described; if it does, it must provide users a means to enable or disable the behavior described.]

validity constraint

[Definition: A rule which applies to all valid XML documents. Violations of validity constraints are errors; they must, at user option, be reported by validating XML processors.]

well-formedness constraint

[Definition: A rule which applies to all well-formed XML documents. Violations of well-formedness constraints are fatal errors.]

match

[Definition: (Of strings or names:) Two strings or names being compared must be identical. Characters with multiple possible representations in ISO/IEC 10646 (e.g. characters with both precomposed and base+diacritic forms) match only if they have the same representation in both strings. No case folding is performed. (Of strings and rules in the grammar:) A string matches a grammatical production if it belongs to the language generated by that production. (Of content and content models:) An element matches its declaration when it conforms in the fashion described in the constraint **[VC: Element Valid]**.]

for compatibility

[Definition: Marks a sentence describing a feature of XML included solely to ensure that XML remains compatible with SGML.]

for interoperability

[Definition: Marks a sentence describing a non-binding recommendation included to increase the chances that XML documents can be processed by the existing installed base of SGML processors which predate the WebSGML Adaptations Annex to ISO 8879.]

2 Documents

[Definition: A data object is an **XML document** if it is well-formed, as defined in this specification. A well-formed XML document may in addition be valid if it meets certain further constraints.]

Each XML document has both a logical and a physical structure. Physically, the document is composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. A document begins in a "root" or document entity. Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup. The logical and physical structures must nest properly, as described in **4.3.2 Well-Formed Parsed Entities**.

2.1 Well-Formed XML Documents

[Definition: A textual object is a **well-formed** XML document if:]

1. Taken as a whole, it matches the production labeled document.
2. It meets all the well-formedness constraints given in this specification.
3. Each of the parsed entities which is referenced directly or indirectly within the document is well-formed.

Document

```
[1] document ::= prolog element Misc*
```

Matching the document production implies that:

1. It contains one or more elements.

2. [Definition: There is exactly one element, called the **root**, or document element, no part of which appears in the content of any other element.] For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. More simply stated, the elements, delimited by start- and end-tags, nest properly within each other.

[Definition: As a consequence of this, for each non-root element `C` in the document, there is one other element `P` in the document such that `C` is in the content of `P`, but is not in the content of any other element that is in the content of `P`. `P` is referred to as the **parent** of `C`, and `C` as a **child** of `P`.]

2.2 Characters

[Definition: A parsed entity contains **text**, a sequence of characters, which may represent markup or character data.] [Definition: A **character** is an atomic unit of text as specified by ISO/IEC 10646 [ISO/IEC 10646] (see also [ISO/IEC 10646-2000]). Legal characters are tab, carriage return, line feed, and the legal characters of Unicode and ISO/IEC 10646. The versions of these standards cited in **A.1 Normative References** were current at the time this document was prepared. New characters may be added to these standards by amendments or new editions. Consequently, XML processors must accept any character in the range specified for Char. The use of "compatibility characters", as defined in section 6.8 of [Unicode] (see also D21 in section 3.6 of [Unicode3]), is discouraged.]

Character Range

```
[2]  Char  ::=  #x9 | #xA | #xD | [#x20-#xD7FF] |
                [#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

/ any Unicode character, excluding the surrogate blocks, FFFE, and FFFF. */*

The mechanism for encoding character code points into bit patterns may vary from entity to entity. All XML processors must accept the UTF-8 and UTF-16 encodings of 10646; the mechanisms for signaling which of the two is in use, or for bringing other encodings into play, are discussed later, in **4.3.3 Character Encoding in Entities**.

2.3 Common Syntactic Constructs

This section defines some symbols used widely in the grammar.

S (white space) consists of one or more space (#x20) characters, carriage returns, line feeds, or tabs.

White Space

```
[3]  S  ::=  (#x20 | #x9 | #xD | #xA)+
```

Characters are classified for convenience as letters, digits, or other characters. A letter consists of an alphabetic or syllabic base character or an ideographic character. Full definitions of the specific characters in each class are given in **B Character Classes**.

[Definition: A **Name** is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters.] Names beginning with the string "`xml`", or any string which would match `(('X'|'x') ('M'|'m') ('L'|'l'))`, are reserved for standardization in this or future versions of this specification.

Note:

The Namespaces in XML Recommendation [XML Names] assigns a meaning to names containing colon characters. Therefore, authors should not use the colon in XML names except for namespace purposes, but XML processors must accept the colon as a name character.

An Nmtoken (name token) is any mixture of name characters.

Names and Tokens

```
[4]  NameChar  ::=  Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar |
                    Extender
[5]  Name      ::=  (Letter | '_' | ':') (NameChar)*
[6]  Names     ::=  Name (S Name)*
[7]  Nmtoken   ::=  (NameChar)+
[8]  Nmtokens  ::=  Nmtoken (S Nmtoken)*
```

Literal data is any quoted string not containing the quotation mark used as a delimiter for that string. Literals are used for specifying the content of internal entities (EntityValue), the values of attributes (AttValue), and external identifiers (SystemLiteral). Note that a SystemLiteral can be parsed without scanning for markup.

Literals

```
[9]   EntityValue    ::=  '"' ([^%&"] | PEReference | Reference)* '"'
                        |  "'" ([^%&'] | PEReference | Reference)* "'"
[10]  AttValue       ::=  '"' ([^<&"] | Reference)* '"'
                        |  "'" ([^<&'] | Reference)* "'"
[11]  SystemLiteral  ::=  ('"' [^"]* '"') | ("'" [^']* "'")
[12]  PubidLiteral   ::=  '"' PubidChar* '"' | "'" (PubidChar - "'")* "'"
[13]  PubidChar      ::=  #x20 | #xD | #xA | [a-zA-Z0-9] | [-'()+,./:=?;!*#@$_%]
```

Note:

Although the EntityValue production allows the definition of an entity consisting of a single explicit < in the literal (e.g., `<!ENTITY mylt "<">`), it is strongly advised to avoid this practice since any reference to that entity will cause a well-formedness error.

2.4 Character Data and Markup

Text consists of intermingled character data and markup. [Definition: **Markup** takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, processing instructions, XML declarations, text declarations, and any white space that is at the top level of the document entity (that is, outside the document element and not inside any other markup).]

[Definition: All text that is not markup constitutes the **character data** of the document.]

The ampersand character (&) and the left angle bracket (<) may appear in their literal form *only* when used as markup delimiters, or within a comment, a processing instruction, or a CDATA section. If they are needed elsewhere, they must be escaped using either numeric character references or the strings "`&`" and "`<`" respectively. The right angle bracket (>) may be represented using the string "`>`", and must, for compatibility, be escaped using "`>`" or a character reference when it appears in the string "`]]>`" in content, when that string is not marking the end of a CDATA section.

In the content of elements, character data is any string of characters which does not contain the start-delimiter of any markup. In a CDATA section, character data is any string of characters not including the CDATA-section-close delimiter, "`]]>`".

To allow attribute values to contain both single and double quotes, the apostrophe or single-quote character (') may be represented as "`'`", and the double-quote character (") as "`"`".

Character Data

```
[14]  CharData  ::=  [^<&]* - ([^<&]* ']]>' [^<&]*)
```

2.5 Comments

[Definition: **Comments** may appear anywhere in a document outside other markup; in addition, they may appear within the document type declaration at places allowed by the grammar. They are not part of the document's character data; an XML processor may, but need not, make it possible for an application to retrieve the text of comments. For compatibility, the string "`--`" (double-hyphen) must not occur within comments.] Parameter entity references are not recognized within comments.

Comments

```
[15] Comment ::= '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
```

An example of a comment:

```
<!-- declarations for <head> & <body> -->
```

Note that the grammar does not allow a comment ending in `--->`. The following example is *not* well-formed.

```
<!-- B+, B, or B--->
```

2.6 Processing Instructions

[Definition: **Processing instructions** (PIs) allow documents to contain instructions for applications.]

Processing Instructions

```
[16] PI       ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
[17] PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
```

PIs are not part of the document's character data, but must be passed through to the application. The PI begins with a target (PITarget) used to identify the application to which the instruction is directed. The target names "`XML`", "`xml`", and so on are reserved for standardization in this or future versions of this specification. The XML Notation mechanism may be used for formal declaration of PI targets. Parameter entity references are not recognized within processing instructions.

2.7 CDATA Sections

[Definition: **CDATA sections** may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string "`<![CDATA[`" and end with the string "`]]>`":]

CDATA Sections

```
[18] CDSect  ::= CDStart CData CDEnd
[19] CDStart ::= '<![CDATA['
[20] CData   ::= (Char* - (Char* ']]>' Char*))
[21] CDEnd   ::= ']]>'
```

Within a CDATA section, only the CDEnd string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form; they need not (and cannot) be escaped using "`<`" and "`&`". CDATA sections cannot nest.

An example of a CDATA section, in which "`<greeting>`" and "`</greeting>`" are recognized as character data, not markup:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

2.8 Prolog and Document Type Declaration

[Definition: XML documents should begin with an **XML declaration** which specifies the version of XML being used.] For example, the following is a complete XML document, well-formed but not valid:

```
<?xml version="1.0"?> <greeting>Hello, world!</greeting>
```

and so is this:

```
<greeting>Hello, world!</greeting>
```

The version number "`1.0`" should be used to indicate conformance to this version of this specification; it is an error for a document to use the value "`1.0`" if it does not conform to this version of this specification. It is the intent of the XML working group to give later versions of this specification numbers other than "`1.0`", but this intent does not indicate a commitment to produce any future versions of XML, nor if any are produced, to use any particular numbering scheme. Since future versions are not ruled out, this construct is provided as a means to allow the possibility of automatic version recognition, should it become necessary. Processors may signal an error if they receive documents labeled with versions they do not support.

The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute-value pairs with its logical structures. XML provides a mechanism, the document type declaration, to define constraints on the logical structure and to support the use of predefined storage units. [Definition: An XML document is **valid** if it has an associated document type declaration and if the document complies with the constraints expressed in it.]

The document type declaration must appear before the first element in the document.

Prolog

```
[22]  prolog        ::=  XMLDecl? Misc* (doctypedecl Misc*)?
[23]  XMLDecl       ::=  '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24]  VersionInfo   ::=  S 'version' Eq ("'" VersionNum "'" | '"' VersionNum
                         '"')/* */
[25]  Eq            ::=  S? '=' S?
[26]  VersionNum    ::=  ([a-zA-Z0-9_.:] | '-')+
[27]  Misc          ::=  Comment | PI | S
```

[Definition: The XML **document type declaration** contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or **DTD**. The document type declaration can point to an external subset (a special kind of external entity) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together.]

[Definition: A **markup declaration** is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration.] These declarations may be contained in whole or in part within parameter entities, as described in the well-formedness and validity constraints below. For further information, see **4 Physical Structures**.

Document Type Definition

[28]	doctypedecl	::=	'<!DOCTYPE' S Name (S ExternalID)? S? ('[' (markupdecl | DeclSep)* ']' S?)? '>'	[VC: Root Element Type] [WFC: External Subset] /* */
[28a]	DeclSep	::=	PEReference | S	[WFC: PE Between Declarations] /* */
[29]	markupdecl	::=	elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment	[VC: Proper Declaration/PE Nesting] [WFC: PEs in Internal Subset]

Note that it is possible to construct a well-formed document containing a doctypedecl that neither points to an external subset nor contains an internal subset.

The markup declarations may be made up in whole or in part of the replacement text of parameter entities. The productions later in this specification for individual nonterminals (elementdecl, AttlistDecl, and so on) describe the declarations *after* all the parameter entities have been included.

Parameter entity references are recognized anywhere in the DTD (internal and external subsets and external parameter entities), except in literals, processing instructions, comments, and the contents of ignored conditional sections (see **3.4 Conditional Sections**). They are also recognized in entity value literals. The use of parameter entities in the internal subset is restricted as described below.

Validity constraint: Root Element Type

The Name in the document type declaration must match the element type of the root element.

Validity constraint: Proper Declaration/PE Nesting

Parameter-entity replacement text must be properly nested with markup declarations. That is to say, if either the first character or the last character of a markup declaration (markupdecl above) is contained in the replacement text for a parameter-entity reference, both must be contained in the same replacement text.

Well-formedness constraint: PEs in Internal Subset

In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.)

Well-formedness constraint: External Subset

The external subset, if any, must match the production for extSubset.

Well-formedness constraint: PE Between Declarations

The replacement text of a parameter entity reference in a DeclSep must match the production extSubsetDecl.

Like the internal subset, the external subset and any external parameter entities referenced in a DeclSep must consist of a series of complete markup declarations of the types allowed by the non-terminal symbol markupdecl, interspersed with white space or parameter-entity references. However, portions of the contents of the external subset or of these external parameter entities may conditionally be ignored by using the conditional section construct; this is not allowed in the internal subset.

External Subset

```
[30] extSubset      ::= TextDecl? extSubsetDecl
[31] extSubsetDecl  ::= ( markupdecl | conditionalSect | DeclSep)* /* */
```

The external subset and external parameter entities also differ from the internal subset in that in them, parameter-entity references are permitted *within* markup declarations, not only *between* markup declarations.

An example of an XML document with a document type declaration:

```
<?xml version="1.0"?> <!DOCTYPE greeting SYSTEM "hello.dtd"> <greeting>Hello,
world!</greeting>
```

The system identifier "`hello.dtd`" gives the address (a URI reference) of a DTD for the document.

The declarations can also be given locally, as in this example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

If both the external and internal subsets are used, the internal subset is considered to occur before the external subset. This has the effect that entity and attribute-list declarations in the internal subset take precedence over those in the external subset.

2.9 Standalone Document Declaration

Markup declarations can affect the content of the document, as passed from an XML processor to an application; examples are attribute defaults and entity declarations. The standalone document declaration, which may appear as a component of the XML declaration, signals whether or not there are such declarations which appear external to the document entity or in parameter entities. [Definition: An **external markup declaration** is defined as a markup declaration occurring in the external subset or in a parameter entity (external or internal, the latter being included because non-validating processors are not required to read them).]

Standalone Document Declaration

```
[32] SDDecl ::= S 'standalone' Eq (("'" ('yes' |
                'no') "'") | ('"' ('yes' | 'no')
                '"'))
```

[VC: Standalone Document Declaration]

In a standalone document declaration, the value "yes" indicates that there are no external markup declarations which affect the information passed from the XML processor to the application. The value "no" indicates that there are or may be such external markup declarations. Note that the standalone document declaration only denotes the presence of external *declarations*; the presence, in a document, of references to external *entities*, when those entities are internally declared, does not change its standalone status.

If there are no external markup declarations, the standalone document declaration has no meaning. If there are external markup declarations but there is no standalone document declaration, the value "no" is assumed.

Any XML document for which `standalone="no"` holds can be converted algorithmically to a standalone document, which may be desirable for some network delivery applications.

Validity constraint: Standalone Document Declaration

The standalone document declaration must have the value "no" if any external markup declarations contain declarations of:

- attributes with default values, if elements to which these attributes apply appear in the document without

specifications of values for these attributes, or

- entities (other than `amp`, `lt`, `gt`, `apos`, `quot`), if references to those entities appear in the document, or
- attributes with values subject to normalization, where the attribute appears in the document with a value which will change as a result of normalization, or
- element types with element content, if white space occurs directly within any instance of those types.

An example XML declaration with a standalone document declaration:

```
<?xml version="1.0" standalone='yes'?>
```

2.10 White Space Handling

In editing XML documents, it is often convenient to use "white space" (spaces, tabs, and blank lines) to set apart the markup for greater readability. Such white space is typically not intended for inclusion in the delivered version of the document. On the other hand, "significant" white space that should be preserved in the delivered version is common, for example in poetry and source code.

An XML processor must always pass all characters in a document that are not markup through to the application. A validating XML processor must also inform the application which of these characters constitute white space appearing in element content.

A special attribute named `xml:space` may be attached to an element to signal an intention that in that element, white space should be preserved by applications. In valid documents, this attribute, like any other, must be declared if it is used. When declared, it must be given as an enumerated type whose values are one or both of "default" and "preserve". For example:

```
<!ATTLIST poem  xml:space (default|preserve) 'preserve'>

<!-- -->
<!ATTLIST pre xml:space (preserve) #FIXED 'preserve'>
```

The value "default" signals that applications' default white-space processing modes are acceptable for this element; the value "preserve" indicates the intent that applications preserve all the white space. This declared intent is considered to apply to all elements within the content of the element where it is specified, unless overriden with another instance of the `xml:space` attribute.

The root element of any document is considered to have signaled no intentions as regards application space handling, unless it provides a value for this attribute or the attribute is declared with a default value.

2.11 End-of-Line Handling

XML parsed entities are often stored in computer files which, for editing convenience, are organized into lines. These lines are typically separated by some combination of the characters carriage-return (#xD) and line-feed (#xA).

To simplify the tasks of applications, the characters passed to an application by the XML processor must be as if the XML processor normalized all line breaks in external parsed entities (including the document entity) on input, before parsing, by translating both the two-character sequence #xD #xA and any #xD that is not followed by #xA to a single #xA character.

2.12 Language Identification

In document processing, it is often useful to identify the natural or formal language in which the content is written. A special attribute named `xml:lang` may be inserted in documents to specify the language used in the contents and attribute values of any element in an XML document. In valid documents, this attribute, like any other, must be

declared if it is used. The values of the attribute are language identifiers as defined by [IETF RFC 1766], Tags for the Identification of Languages, or its successor on the IETF Standards Track.

Note:

[IETF RFC 1766] tags are constructed from two-letter language codes as defined by [ISO 639], from two-letter country codes as defined by [ISO 3166], or from language identifiers registered with the Internet Assigned Numbers Authority [IANA-LANGCODES]. It is expected that the successor to [IETF RFC 1766] will introduce three-letter language codes for languages not presently covered by [ISO 639].

(Productions 33 through 38 have been removed.)

For example:

```
<p xml:lang="en">The quick brown fox jumps over the lazy dog.</p>
<p xml:lang="en-GB">What colour is it?</p>
<p xml:lang="en-US">What color is it?</p>
<sp who="Faust" desc='leise' xml:lang="de">
  <l>Habe nun, ach! Philosophie,</l>
  <l>Juristerei, und Medizin</l>
  <l>und leider auch Theologie</l>
  <l>durchaus studiert mit heißem Bemüh'n.</l>
</sp>
```

The intent declared with `xml:lang` is considered to apply to all attributes and content of the element where it is specified, unless overridden with an instance of `xml:lang` on another element within that content.

A simple declaration for `xml:lang` might take the form

```
xml:lang NMTOKEN #IMPLIED
```

but specific default values may also be given, if appropriate. In a collection of French poems for English students, with glosses and notes in English, the `xml:lang` attribute might be declared this way:

```
<!ATTLIST poem   xml:lang NMTOKEN 'fr'>
<!ATTLIST gloss  xml:lang NMTOKEN 'en'>
<!ATTLIST note   xml:lang NMTOKEN 'en'>
```

3 Logical Structures

[Definition: Each XML document contains one or more **elements**, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications.] Each attribute specification has a name and a value.

Element

[39] `element` ::= `EmptyElemTag`
| `STag` `content` `ETag` [WFC: Element Type Match]
[VC: Element Valid]

This specification does not constrain the semantics, use, or (beyond syntax) names of the element types and attributes, except that names beginning with a match to `(('X'|'x')('M'|'m')('L'|'l'))` are reserved for standardization in this or future versions of this specification.

Well-formedness constraint: Element Type Match

The Name in an element's end-tag must match the element type in the start-tag.

Validity constraint: Element Valid

An element is valid if there is a declaration matching elementdecl where the Name matches the element type, and one of the following holds:

1. The declaration matches **EMPTY** and the element has no content.
2. The declaration matches children and the sequence of child elements belongs to the language generated by the regular expression in the content model, with optional white space (characters matching the nonterminal S) between the start-tag and the first child element, between child elements, or between the last child element and the end-tag. Note that a CDATA section containing only white space does not match the nonterminal S, and hence cannot appear in these positions.
3. The declaration matches Mixed and the content consists of character data and child elements whose types match names in the content model.
4. The declaration matches **ANY**, and the types of any child elements have been declared.

3.1 Start-Tags, End-Tags, and Empty-Element Tags

[Definition: The beginning of every non-empty XML element is marked by a **start-tag**.]

Start-tag

```
[40] STag      ::= '<' Name (S Attribute)* S? '>' [WFC: Unique Att Spec]
[41] Attribute ::= Name Eq AttValue               [VC: Attribute Value Type]
                                                  [WFC: No External Entity References]
                                                  [WFC: No < in Attribute Values]
```

The Name in the start- and end-tags gives the element's **type**. [Definition: The Name-AttValue pairs are referred to as the **attribute specifications** of the element], [Definition: with the Name in each pair referred to as the **attribute name**] and [Definition: the content of the AttValue (the text between the ' or " delimiters) as the **attribute value**.]Note that the order of attribute specifications in a start-tag or empty-element tag is not significant.

Well-formedness constraint: Unique Att Spec

No attribute name may appear more than once in the same start-tag or empty-element tag.

Validity constraint: Attribute Value Type

The attribute must have been declared; the value must be of the type declared for it. (For attribute types, see **3.3 Attribute-List Declarations**.)

Well-formedness constraint: No External Entity References

Attribute values cannot contain direct or indirect entity references to external entities.

Well-formedness constraint: No < in Attribute Values

The replacement text of any entity referred to directly or indirectly in an attribute value must not contain a <.

An example of a start-tag:

```
<termdef id="dt-dog" term="dog">
```

[Definition: The end of every element that begins with a start-tag must be marked by an **end-tag** containing a name that echoes the element's type as given in the start-tag:]

End-tag

```
[42]  ETag  ::=  '</' Name S? '>'
```

An example of an end-tag:

```
</termdef>
```

[Definition: The text between the start-tag and end-tag is called the element's **content**:]

Content of Elements

```
[43]  content  ::=  CharData? ((element | Reference | CDSect | PI | Comment)   /* */
                    CharData?)*
```

[Definition: An element with no content is said to be **empty**.] The representation of an empty element is either a start-tag immediately followed by an end-tag, or an empty-element tag. [Definition: An **empty-element tag** takes a special form:]

Tags for Empty Elements

```
[44]  EmptyElemTag  ::=  '<' Name (S Attribute)* S? '/>' [WFC: Unique Att Spec]
```

Empty-element tags may be used for any element which has no content, whether or not it is declared using the keyword **EMPTY**. For interoperability, the empty-element tag should be used, and should only be used, for elements which are declared EMPTY.

Examples of empty elements:

```
<IMG align="left"
 src="http://www.w3.org/Icons/WWW/w3c_home" />
<br></br>
<br/>
```

3.2 Element Type Declarations

The element structure of an XML document may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element's content.

Element type declarations often constrain which element types can appear as children of the element. At user option, an XML processor may issue a warning when a declaration mentions an element type for which no declaration is provided, but this is not an error.

[Definition: An **element type declaration** takes the form:]

Element Type Declaration

```
[45]  elementdecl  ::=  '<!ELEMENT' S Name S contentspec S? '>'   [VC: Unique Element Type Declaration]
[46]  contentspec  ::=  'EMPTY' | 'ANY' | Mixed | children
```

where the Name gives the element type being declared.

Validity constraint: Unique Element Type Declaration

No element type may be declared more than once.

Examples of element type declarations:

```
<!ELEMENT br EMPTY>
<!ELEMENT p (#PCDATA|emph)* >
<!ELEMENT %name.para; %content.para; >
<!ELEMENT container ANY>
```

3.2.1 Element Content

[Definition: An element type has **element content** when elements of that type must contain only child elements (no character data), optionally separated by white space (characters matching the nonterminal S).][Definition: In this case, the constraint includes a **content model**, a simple grammar governing the allowed types of the child elements and the order in which they are allowed to appear.] The grammar is built on content particles (cps), which consist of names, choice lists of content particles, or sequence lists of content particles:

Element-content Models

```
[47]  children  ::=  (choice | seq) ('?' | '*' | '+')?
[48]  cp        ::=  (Name | choice | seq) ('?' | '*' |
                     '+')?
[49]  choice    ::=  '(' S? cp ( S? '|' S? cp )+ S? ')'      /* */
                                                             /* */
                                                             [VC: Proper Group/PE Nesting]
[50]  seq       ::=  '(' S? cp ( S? ',' S? cp )* S? ')'      /* */
                                                             [VC: Proper Group/PE Nesting]
```

where each Name is the type of an element which may appear as a child. Any content particle in a choice list may appear in the element content at the location where the choice list appears in the grammar; content particles occurring in a sequence list must each appear in the element content in the order given in the list. The optional character following a name or list governs whether the element or the content particles in the list may occur one or more (+), zero or more (*), or zero or one times (?). The absence of such an operator means that the element or content particle must appear exactly once. This syntax and meaning are identical to those used in the productions in this specification.

The content of an element matches a content model if and only if it is possible to trace out a path through the content model, obeying the sequence, choice, and repetition operators and matching each element in the content against an element type in the content model. For compatibility, it is an error if an element in the document can match more than one occurrence of an element type in the content model. For more information, see **E Deterministic Content Models**.

Validity constraint: Proper Group/PE Nesting

Parameter-entity replacement text must be properly nested with parenthesized groups. That is to say, if either of the opening or closing parentheses in a choice, seq, or Mixed construct is contained in the replacement text for a parameter entity, both must be contained in the same replacement text.

For interoperability, if a parameter-entity reference appears in a choice, seq, or Mixed construct, its replacement text should contain at least one non-blank character, and neither the first nor last non-blank character of the replacement text should be a connector (| or ,).

Examples of element-content models:

```
<!ELEMENT spec (front, body, back?)>
<!ELEMENT div1 (head, (p | list | note)*, div2*)>
<!ELEMENT dictionary-body (%div.mix; | %dict.mix;)*>
```

3.2.2 Mixed Content

[Definition: An element type has **mixed content** when elements of that type may contain character data, optionally interspersed with child elements.] In this case, the types of the child elements may be constrained, but not their order or their number of occurrences:

Mixed-content Declaration

```
[51] Mixed ::= '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
             | '(' S? '#PCDATA' S? ')'                    [VC: Proper Group/PE Nesting]
                                                          [VC: No Duplicate Types]
```

where the Names give the types of elements that may appear as children. The keyword **#PCDATA** derives historically from the term "parsed character data."

Validity constraint: No Duplicate Types

The same name must not appear more than once in a single mixed-content declaration.

Examples of mixed content declarations:

```
<!ELEMENT p (#PCDATA|a|ul|b|i|em)*>
<!ELEMENT p (#PCDATA | %font; | %phrase; | %special; | %form;)* >
<!ELEMENT b (#PCDATA)>
```

3.3 Attribute-List Declarations

Attributes are used to associate name-value pairs with elements. Attribute specifications may appear only within start-tags and empty-element tags; thus, the productions used to recognize them appear in **3.1 Start-Tags, End-Tags, and Empty-Element Tags**. Attribute-list declarations may be used:

- To define the set of attributes pertaining to a given element type.
- To establish type constraints for these attributes.
- To provide default values for attributes.

[Definition: **Attribute-list declarations** specify the name, data type, and default value (if any) of each attribute associated with a given element type:]

Attribute-list Declaration

```
[52] AttlistDecl ::= '<!ATTLIST' S Name AttDef* S? '>'
[53] AttDef      ::= S Name S AttType S DefaultDecl
```

The Name in the AttlistDecl rule is the type of an element. At user option, an XML processor may issue a warning if attributes are declared for an element type not itself declared, but this is not an error. The Name in the AttDef rule is the name of the attribute.

When more than one AttlistDecl is provided for a given element type, the contents of all those provided are merged. When more than one definition is provided for the same attribute of a given element type, the first declaration is binding and later declarations are ignored. For interoperability, writers of DTDs may choose to provide at most one

attribute-list declaration for a given element type, at most one attribute definition for a given attribute name in an attribute-list declaration, and at least one attribute definition in each attribute-list declaration. For interoperability, an XML processor may at user option issue a warning when more than one attribute-list declaration is provided for a given element type, or more than one attribute definition is provided for a given attribute, but this is not an error.

3.3.1 Attribute Types

XML attribute types are of three kinds: a string type, a set of tokenized types, and enumerated types. The string type may take any literal string as a value; the tokenized types have varying lexical and semantic constraints. The validity constraints noted in the grammar are applied after the attribute value has been normalized as described in **3.3 Attribute-List Declarations**.

Attribute Types

```
[54]  AttType         ::=  StringType | TokenizedType |
                           EnumeratedType
[55]  StringType      ::=  'CDATA'
[56]  TokenizedType   ::=  'ID'                          [VC: ID]
                                                         [VC: One ID per
                                                         Element Type]
                                                         [VC: ID Attribute
                                                         Default]
                           | 'IDREF'                     [VC: IDREF]
                           | 'IDREFS'                    [VC: IDREF]
                           | 'ENTITY'                    [VC: Entity Name]
                           | 'ENTITIES'                  [VC: Entity Name]
                           | 'NMTOKEN'                   [VC: Name Token]
                           | 'NMTOKENS'                  [VC: Name Token]
```

Validity constraint: ID

Values of type **ID** must match the Name production. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must uniquely identify the elements which bear them.

Validity constraint: One ID per Element Type

No element type may have more than one ID attribute specified.

Validity constraint: ID Attribute Default

An ID attribute must have a declared default of **#IMPLIED** or **#REQUIRED**.

Validity constraint: IDREF

Values of type **IDREF** must match the Name production, and values of type **IDREFS** must match Names; each Name must match the value of an ID attribute on some element in the XML document; i.e. **IDREF** values must match the value of some ID attribute.

Validity constraint: Entity Name

Values of type **ENTITY** must match the Name production, values of type **ENTITIES** must match Names; each Name must match the name of an unparsed entity declared in the DTD.

Validity constraint: Name Token

Values of type **NMTOKEN** must match the Nmtoken production; values of type **NMTOKENS** must match

Nmtokens.

[Definition: **Enumerated attributes** can take one of a list of values provided in the declaration]. There are two kinds of enumerated types:

Enumerated Attribute Types

[57]	EnumeratedType	::=	NotationType | Enumeration	
[58]	NotationType	::=	'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'	[VC: Notation Attributes] [VC: One Notation Per Element Type] [VC: No Notation on Empty Element]
[59]	Enumeration	::=	'(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'	[VC: Enumeration]

A **NOTATION** attribute identifies a notation, declared in the DTD with associated system and/or public identifiers, to be used in interpreting the element to which the attribute is attached.

Validity constraint: Notation Attributes

Values of this type must match one of the notation names included in the declaration; all notation names in the declaration must be declared.

Validity constraint: One Notation Per Element Type

No element type may have more than one **NOTATION** attribute specified.

Validity constraint: No Notation on Empty Element

For compatibility, an attribute of type **NOTATION** must not be declared on an element declared **EMPTY**.

Validity constraint: Enumeration

Values of this type must match one of the Nmtoken tokens in the declaration.

For interoperability, the same Nmtoken should not occur more than once in the enumerated attribute types of a single element type.

3.3.2 Attribute Defaults

An attribute declaration provides information on whether the attribute's presence is required, and if not, how an XML processor should react if a declared attribute is absent in a document.

Attribute Defaults

[60]	DefaultDecl	::=	'#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)	[VC: Required Attribute] [VC: Attribute Default Legal] [WFC: No < in Attribute Values] [VC: Fixed Attribute Default]

In an attribute declaration, **#REQUIRED** means that the attribute must always be provided, **#IMPLIED** that no default value is provided. [Definition: If the declaration is neither **#REQUIRED** nor **#IMPLIED**, then the AttValue value contains the declared **default** value; the **#FIXED** keyword states that the attribute must always have the default value. If a default value is declared, when an XML processor encounters an omitted attribute, it is to behave as though

the attribute were present with the declared default value.]

Validity constraint: Required Attribute

If the default declaration is the keyword **#REQUIRED**, then the attribute must be specified for all elements of the type in the attribute-list declaration.

Validity constraint: Attribute Default Legal

The declared default value must meet the lexical constraints of the declared attribute type.

Validity constraint: Fixed Attribute Default

If an attribute has a default value declared with the **#FIXED** keyword, instances of that attribute must match the default value.

Examples of attribute-list declarations:

```
<!ATTLIST termdef
          id      ID      #REQUIRED
          name    CDATA   #IMPLIED>
<!ATTLIST list
          type    (bullets|ordered|glossary)  "ordered">
<!ATTLIST form
          method  CDATA   #FIXED "POST">
```

3.3.3 Attribute-Value Normalization

Before the value of an attribute is passed to the application or checked for validity, the XML processor must normalize the attribute value by applying the algorithm below, or by using some other method such that the value passed to the application is the same as that produced by the algorithm.

1. All line breaks must have been normalized on input to #xA as described in **2.11 End-of-Line Handling**, so the rest of this algorithm operates on text normalized in this way.
2. Begin with a normalized value consisting of the empty string.
3. For each character, entity reference, or character reference in the unnormalized attribute value, beginning with the first and continuing to the last, do the following:
   - For a character reference, append the referenced character to the normalized value.
   - For an entity reference, recursively apply step 3 of this algorithm to the replacement text of the entity.
   - For a white space character (#x20, #xD, #xA, #x9), append a space character (#x20) to the normalized value.
   - For another character, append the character to the normalized value.

If the attribute type is not CDATA, then the XML processor must further process the normalized attribute value by discarding any leading and trailing space (#x20) characters, and by replacing sequences of space (#x20) characters by a single space (#x20) character.

Note that if the unnormalized attribute value contains a character reference to a white space character other than space (#x20), the normalized value contains the referenced character itself (#xD, #xA or #x9). This contrasts with the case where the unnormalized value contains a white space character (not a reference), which is replaced with a space character (#x20) in the normalized value and also contrasts with the case where the unnormalized value contains an entity reference whose replacement text contains a white space character; being recursively processed, the white space character is replaced with a space character (#x20) in the normalized value.

All attributes for which no declaration has been read should be treated by a non-validating processor as if declared **CDATA**.

Following are examples of attribute normalization. Given the following declarations:

```
<!ENTITY d "&#xD;">
<!ENTITY a "&#xA;">
<!ENTITY da "&#xD;&#xA;">
```

the attribute specifications in the left column below would be normalized to the character sequences of the middle column if the attribute a is declared **NMTOKENS** and to those of the right columns if a is declared **CDATA**.

Attribute specification	a is NMTOKENS	a is CDATA
a=" xyz"	x y z	#x20 #x20 x y z
a="&d;&d;A&a;&a;B&da;"	A #x20 B	#x20 #x20 A #x20 #x20 B #x20 #x20
a= "

A

B
" | #xD #xD A #xA #xA B #xD
#xA | #xD #xD A #xA
#xA B #xD #xD |

Note that the last example is invalid (but well-formed) if a is declared to be of type **NMTOKENS**.

3.4 Conditional Sections

[Definition: **Conditional sections** are portions of the document type declaration external subset which are included in, or excluded from, the logical structure of the DTD based on the keyword which governs them.]

Conditional Section

```
[61] conditionalSect    ::= includeSect | ignoreSect
[62] includeSect        ::= '<![' S? 'INCLUDE' S? '['
                            extSubsetDecl ']]>'          /* */
                                                         [VC: Proper
                                                         Conditional
                                                         Section/PE Nesting]
[63] ignoreSect         ::= '<![' S? 'IGNORE' S? '['
                            ignoreSectContents* ']]>'    /* */
                                                         [VC: Proper
                                                         Conditional
                                                         Section/PE Nesting]
[64] ignoreSectContents ::= Ignore ('<![' ignoreSectContents
                            ']]>' Ignore)*
[65] Ignore             ::= Char* - (Char* ('<![' | ']]>')
                            Char*)
```

Validity constraint: Proper Conditional Section/PE Nesting

If any of the "<![", "[", or "]]>" of a conditional section is contained in the replacement text for a parameter-entity reference, all of them must be contained in the same replacement text.

Like the internal and external DTD subsets, a conditional section may contain one or more complete declarations, comments, processing instructions, or nested conditional sections, intermingled with white space.

If the keyword of the conditional section is **INCLUDE**, then the contents of the conditional section are part of the DTD. If the keyword of the conditional section is **IGNORE**, then the contents of the conditional section are not logically part of the DTD. If a conditional section with a keyword of **INCLUDE** occurs within a larger conditional section with a keyword of **IGNORE**, both the outer and the inner conditional sections are ignored. The contents of an ignored conditional section are parsed by ignoring all characters after the "[" following the keyword, except conditional section starts "<![" and ends "]]>", until the matching conditional section end is found. Parameter entity references are not recognized in this process.

If the keyword of the conditional section is a parameter-entity reference, the parameter entity must be replaced by its content before the processor decides whether to include or ignore the conditional section.

An example:

```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >

<![%draft;[
<!ELEMENT book (comments*, title, body, supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>
```

4 Physical Structures

[Definition: An XML document may consist of one or many storage units. These are called **entities**; they all have **content** and are all (except for the document entity and the external DTD subset) identified by entity **name**.] Each XML document has one entity called the document entity, which serves as the starting point for the XML processor and may contain the whole document.

Entities may be either parsed or unparsed. [Definition: A **parsed entity's** contents are referred to as its replacement text; this text is considered an integral part of the document.]

[Definition: An **unparsed entity** is a resource whose contents may or may not be text, and if text, may be other than XML. Each unparsed entity has an associated notation, identified by name. Beyond a requirement that an XML processor make the identifiers for the entity and notation available to the application, XML places no constraints on the contents of unparsed entities.]

Parsed entities are invoked by name using entity references; unparsed entities by name, given in the value of **ENTITY** or **ENTITIES** attributes.

[Definition: **General entities** are entities for use within the document content. In this specification, general entities are sometimes referred to with the unqualified term *entity* when this leads to no ambiguity.] [Definition: **Parameter entities** are parsed entities for use within the DTD.] These two types of entities use different forms of reference and are recognized in different contexts. Furthermore, they occupy different namespaces; a parameter entity and a general entity with the same name are two distinct entities.

4.1 Character and Entity References

[Definition: A **character reference** refers to a specific character in the ISO/IEC 10646 character set, for example one not directly accessible from available input devices.]

Character Reference

```
[66]  CharRef  ::=  '&#' [0-9]+ ';'
                  | '&#x' [0-9a-fA-F]+ ';' [WFC: Legal Character]
```

Well-formedness constraint: Legal Character

Characters referred to using character references must match the production for Char.

If the character reference begins with "&#x", the digits and letters up to the terminating ; provide a hexadecimal representation of the character's code point in ISO/IEC 10646. If it begins just with "&#", the digits up to the terminating ; provide a decimal representation of the character's code point.

[Definition: An **entity reference** refers to the content of a named entity.] [Definition: References to parsed general entities use ampersand (&) and semicolon (;) as delimiters.] [Definition: **Parameter-entity references** use percent-sign (%) and semicolon (;) as delimiters.]

Entity Reference

[67]	Reference	::=	EntityRef | CharRef	
[68]	EntityRef	::=	'&' Name ';'	[WFC: Entity Declared] [VC: Entity Declared] [WFC: Parsed Entity] [WFC: No Recursion]
[69]	PEReference	::=	'%' Name ';'	[VC: Entity Declared] [WFC: No Recursion] [WFC: In DTD]

Well-formedness constraint: Entity Declared

In a document without any DTD, a document with only an internal DTD subset which contains no parameter entity references, or a document with "standalone='yes'", for an entity reference that does not occur within the external subset or a parameter entity, the Name given in the entity reference must match that in an entity declaration that does not occur within the external subset or a parameter entity, except that well-formed documents need not declare any of the following entities: amp, lt, gt, apos, quot. The declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration.

Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be declared is a well-formedness constraint only if standalone='yes'.

Validity constraint: Entity Declared

In a document with an external subset or external parameter entities with "standalone='no'", the Name given in the entity reference must match that in an entity declaration. For interoperability, valid documents should declare the entities amp, lt, gt, apos, quot, in the form specified in **4.6 Predefined Entities**. The declaration of a parameter entity must precede any reference to it. Similarly, the declaration of a general entity must precede any attribute-list declaration containing a default value with a direct or indirect reference to that general entity.

Well-formedness constraint: Parsed Entity

An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type **ENTITY** or **ENTITIES**.

Well-formedness constraint: No Recursion

A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

Well-formedness constraint: In DTD

Parameter-entity references may only appear in the DTD.

Examples of character and entity references:

```
Type <key>less-than</key> (&#x3C;) to save options.
This document was prepared on &docdate; and
is classified &security-level;.
```

Example of a parameter-entity reference:

```
<!-- declare the parameter entity "ISOLat2"... -->
<!ENTITY % ISOLat2
         SYSTEM "http://www.xml.com/iso/isolat2-xml.entities" >
<!-- ... now reference it. -->
%ISOLat2;
```

4.2 Entity Declarations

[Definition: Entities are declared thus:]

Entity Declaration

```
[70]  EntityDecl  ::=  GEDecl | PEDecl
[71]  GEDecl      ::=  '<!ENTITY' S Name S EntityDef S? '>'
[72]  PEDecl      ::=  '<!ENTITY' S '%' S Name S PEDef S? '>'
[73]  EntityDef   ::=  EntityValue | (ExternalID NDataDecl?)
[74]  PEDef       ::=  EntityValue | ExternalID
```

The Name identifies the entity in an entity reference or, in the case of an unparsed entity, in the value of an **ENTITY** or **ENTITIES** attribute. If the same entity is declared more than once, the first declaration encountered is binding; at user option, an XML processor may issue a warning if entities are declared multiple times.

4.2.1 Internal Entities

[Definition: If the entity definition is an EntityValue, the defined entity is called an **internal entity**. There is no separate physical storage object, and the content of the entity is given in the declaration.] Note that some processing of entity and character references in the literal entity value may be required to produce the correct replacement text: see **4.5 Construction of Internal Entity Replacement Text**.

An internal entity is a parsed entity.

Example of an internal entity declaration:

```
<!ENTITY Pub-Status "This is a pre-release of the
 specification.">
```

4.2.2 External Entities

[Definition: If the entity is not internal, it is an **external entity**, declared as follows:]

External Entity Declaration

```
[75]  ExternalID  ::=  'SYSTEM' S SystemLiteral
                     | 'PUBLIC' S PubidLiteral S SystemLiteral
[76]  NDataDecl   ::=  S 'NDATA' S Name                    [VC: Notation Declared]
```

If the NDataDecl is present, this is a general unparsed entity; otherwise it is a parsed entity.

Validity constraint: Notation Declared

The Name must match the declared name of a notation.

[Definition: The SystemLiteral is called the entity's **system identifier**. It is a URI reference (as defined in [IETF RFC 2396], updated by [IETF RFC 2732]), meant to be dereferenced to obtain input for the XML processor to construct the entity's replacement text.] It is an error for a fragment identifier (beginning with a # character) to be part of a system identifier. Unless otherwise provided by information outside the scope of this specification (e.g. a special XML element type defined by a particular DTD, or a processing instruction defined by a particular application specification), relative URIs are relative to the location of the resource within which the entity declaration occurs. A URI might thus be relative to the document entity, to the entity containing the external DTD subset, or to some other external parameter entity.

URI references require encoding and escaping of certain characters. The disallowed characters include all non-ASCII characters, plus the excluded characters listed in Section 2.4 of [IETF RFC 2396], except for the number sign (#) and percent sign (%) characters and the square bracket characters re-allowed in [IETF RFC 2732]. Disallowed characters must be escaped as follows:

1. Each disallowed character is converted to UTF-8 [IETF RFC 2279] as one or more bytes.
2. Any octets corresponding to a disallowed character are escaped with the URI escaping mechanism (that is, converted to %HH, where HH is the hexadecimal notation of the byte value).
3. The original character is replaced by the resulting character sequence.

[Definition: In addition to a system identifier, an external identifier may include a **public identifier**.] An XML processor attempting to retrieve the entity's content may use the public identifier to try to generate an alternative URI reference. If the processor is unable to do so, it must use the URI reference specified in the system literal. Before a match is attempted, all strings of white space in the public identifier must be normalized to single space characters (#x20), and leading and trailing white space must be removed.

Examples of external entity declarations:

```
<!ENTITY open-hatch
         SYSTEM "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY open-hatch
         PUBLIC "-//Textuality//TEXT Standard open-hatch boilerplate//EN"
         "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY hatch-pic
         SYSTEM "../grafix/OpenHatch.gif"
         NDATA gif >
```

4.3 Parsed Entities

4.3.1 The Text Declaration

External parsed entities should each begin with a **text declaration**.

Text Declaration

```
[77]  TextDecl  ::=  '<?xml' VersionInfo? EncodingDecl S? '?>'
```

The text declaration must be provided literally, not by reference to a parsed entity. No text declaration may appear at any position other than the beginning of an external parsed entity. The text declaration in an external parsed entity is not considered part of its replacement text.

4.3.2 Well-Formed Parsed Entities

The document entity is well-formed if it matches the production labeled document. An external general parsed entity is well-formed if it matches the production labeled extParsedEnt. All external parameter entities are well-formed by definition.

Well-Formed External Parsed Entity

```
[78]  extParsedEnt  ::=  TextDecl? content
```

An internal general parsed entity is well-formed if its replacement text matches the production labeled content. All internal parameter entities are well-formed by definition.

A consequence of well-formedness in entities is that the logical and physical structures in an XML document are properly nested; no start-tag, end-tag, empty-element tag, element, comment, processing instruction, character reference, or entity reference can begin in one entity and end in another.

4.3.3 Character Encoding in Entities

Each external parsed entity in an XML document may use a different encoding for its characters. All XML processors must be able to read entities in both the UTF-8 and UTF-16 encodings. The terms "UTF-8" and "UTF-16" in this specification do not apply to character encodings with any other labels, even if the encodings or labels are very similar to UTF-8 or UTF-16.

Entities encoded in UTF-16 must begin with the Byte Order Mark described by Annex F of [ISO/IEC 10646], Annex H of [ISO/IEC 10646-2000], section 2.4 of [Unicode], and section 2.7 of [Unicode3] (the ZERO WIDTH NO-BREAK SPACE character, #xFEFF). This is an encoding signature, not part of either the markup or the character data of the XML document. XML processors must be able to use this character to differentiate between UTF-8 and UTF-16 encoded documents.

Although an XML processor is required to read only entities in the UTF-8 and UTF-16 encodings, it is recognized that other encodings are used around the world, and it may be desired for XML processors to read entities that use them. In the absence of external character encoding information (such as MIME headers), parsed entities which are stored in an encoding other than UTF-8 or UTF-16 must begin with a text declaration (see **4.3.1 The Text Declaration**) containing an encoding declaration:

Encoding Declaration

```
[80]  EncodingDecl  ::=  S 'encoding' Eq ('"' EncName '"' |
                         "'" EncName "'" )
[81]  EncName       ::=  [A-Za-z] ([A-Za-z0-9._] | '-')*      /* Encoding name contains
                                                               only Latin characters */
```

In the document entity, the encoding declaration is part of the XML declaration. The EncName is the name of the encoding used.

In an encoding declaration, the values "`UTF-8`", "`UTF-16`", "`ISO-10646-UCS-2`", and "`ISO-10646-UCS-4`" should be used for the various encodings and transformations of Unicode / ISO/IEC 10646, the values "`ISO-8859-1`", "`ISO-8859-2`", ... "`ISO-8859-`n" (where n is the part number) should be used for the parts of ISO 8859, and the values "`ISO-2022-JP`", "`Shift_JIS`", and "`EUC-JP`" should be used for the various encoded forms of JIS X-0208-1997. It is recommended that character encodings registered (as *charset*s) with the Internet Assigned Numbers Authority [IANA-CHARSETS], other than those just listed, be referred to using their registered names; other encodings should use names starting with an "x-" prefix. XML processors should match character encoding names in a case-insensitive way and should either interpret an IANA-registered name as the encoding registered at IANA for that name or treat it as unknown (processors are, of course, not required to support all IANA-registered encodings).

In the absence of information provided by an external transport protocol (e.g. HTTP or MIME), it is an error for an entity including an encoding declaration to be presented to the XML processor in an encoding other than that named

in the declaration, or for an entity which begins with neither a Byte Order Mark nor an encoding declaration to use an encoding other than UTF-8. Note that since ASCII is a subset of UTF-8, ordinary ASCII entities do not strictly need an encoding declaration.

It is a fatal error for a TextDecl to occur other than at the beginning of an external entity.

It is a fatal error when an XML processor encounters an entity with an encoding that it is unable to process. It is a fatal error if an XML entity is determined (via default, encoding declaration, or higher-level protocol) to be in a certain encoding but contains octet sequences that are not legal in that encoding. It is also a fatal error if an XML entity contains no encoding declaration and its content is not legal UTF-8 or UTF-16.

Examples of text declarations containing encoding declarations:

```
<?xml encoding='UTF-8'?>
<?xml encoding='EUC-JP'?>
```

4.4 XML Processor Treatment of Entities and References

The table below summarizes the contexts in which character references, entity references, and invocations of unparsed entities might appear and the required behavior of an XML processor in each case. The labels in the leftmost column describe the recognition context:

Reference in Content

> as a reference anywhere after the start-tag and before the end-tag of an element; corresponds to the nonterminal content.

Reference in Attribute Value

> as a reference within either the value of an attribute in a start-tag, or a default value in an attribute declaration; corresponds to the nonterminal AttValue.

Occurs as Attribute Value

> as a Name, not a reference, appearing either as the value of an attribute which has been declared as type **ENTITY**, or as one of the space-separated tokens in the value of an attribute which has been declared as type **ENTITIES**.

Reference in Entity Value

> as a reference within a parameter or internal entity's literal entity value in the entity's declaration; corresponds to the nonterminal EntityValue.

Reference in DTD

> as a reference within either the internal or external subsets of the DTD, but outside of an EntityValue, AttValue, PI, Comment, SystemLiteral, PubidLiteral, or the contents of an ignored conditional section (see **3.4 Conditional Sections**).

.

	Entity Type				Character
	Parameter	Internal General	External Parsed General	Unparsed	
Reference in Content	Not recognized	Included	Included if validating	Forbidden	Included
Reference in Attribute Value	Not recognized	Included in literal	Forbidden	Forbidden	Included

Occurs as Attribute Value	Not recognized	Forbidden	Forbidden	Notify	Not recognized
Reference in EntityValue	Included in literal	Bypassed	Bypassed	Forbidden	Included
Reference in DTD	Included as PE	Forbidden	Forbidden	Forbidden	Forbidden

4.4.1 Not Recognized

Outside the DTD, the % character has no special significance; thus, what would be parameter entity references in the DTD are not recognized as markup in content. Similarly, the names of unparsed entities are not recognized except when they appear in the value of an appropriately declared attribute.

4.4.2 Included

[Definition: An entity is **included** when its replacement text is retrieved and processed, in place of the reference itself, as though it were part of the document at the location the reference was recognized.] The replacement text may contain both character data and (except for parameter entities) markup, which must be recognized in the usual way. (The string "`AT&T;`" expands to "`AT&T;`" and the remaining ampersand is not recognized as an entity-reference delimiter.) A character reference is **included** when the indicated character is processed in place of the reference itself.

4.4.3 Included If Validating

When an XML processor recognizes a reference to a parsed entity, in order to validate the document, the processor must include its replacement text. If the entity is external, and the processor is not attempting to validate the XML document, the processor may, but need not, include the entity's replacement text. If a non-validating processor does not include the replacement text, it must inform the application that it recognized, but did not read, the entity.

This rule is based on the recognition that the automatic inclusion provided by the SGML and XML entity mechanism, primarily designed to support modularity in authoring, is not necessarily appropriate for other applications, in particular document browsing. Browsers, for example, when encountering an external parsed entity reference, might choose to provide a visual indication of the entity's presence and retrieve it for display only on demand.

4.4.4 Forbidden

The following are forbidden, and constitute fatal errors:

- the appearance of a reference to an unparsed entity.
- the appearance of any character or general-entity reference in the DTD except within an EntityValue or AttValue.
- a reference to an external entity in an attribute value.

4.4.5 Included in Literal

When an entity reference appears in an attribute value, or a parameter entity reference appears in a literal entity value, its replacement text is processed in place of the reference itself as though it were part of the document at the location the reference was recognized, except that a single or double quote character in the replacement text is always treated as a normal data character and will not terminate the literal. For example, this is well-formed:

```
<!--  -->
<!ENTITY % YN '"Yes"' >
<!ENTITY WhatHeSaid "He said %YN;" >
```

while this is not:

```
<!ENTITY EndAttr "27'" >
<element attribute='a-&EndAttr;>
```

4.4.6 Notify

When the name of an unparsed entity appears as a token in the value of an attribute of declared type **ENTITY** or **ENTITIES**, a validating processor must inform the application of the system and public (if any) identifiers for both the entity and its associated notation.

4.4.7 Bypassed

When a general entity reference appears in the EntityValue in an entity declaration, it is bypassed and left as is.

4.4.8 Included as PE

Just as with external parsed entities, parameter entities need only be included if validating. When a parameter-entity reference is recognized in the DTD and included, its replacement text is enlarged by the attachment of one leading and one following space (#x20) character; the intent is to constrain the replacement text of parameter entities to contain an integral number of grammatical tokens in the DTD. This behavior does not apply to parameter entity references within entity values; these are described in **4.4.5 Included in Literal**.

4.5 Construction of Internal Entity Replacement Text

In discussing the treatment of internal entities, it is useful to distinguish two forms of the entity's value. [Definition: The **literal entity value** is the quoted string actually present in the entity declaration, corresponding to the non-terminal EntityValue.] [Definition: The **replacement text** is the content of the entity, after replacement of character references and parameter-entity references.]

The literal entity value as given in an internal entity declaration (EntityValue) may contain character, parameter-entity, and general-entity references. Such references must be contained entirely within the literal entity value. The actual replacement text that is included as described above must contain the *replacement text* of any parameter entities referred to, and must contain the character referred to, in place of any character references in the literal entity value; however, general-entity references must be left as-is, unexpanded. For example, given the following declarations:

```
<!ENTITY % pub    "&#xc9;ditions Gallimard" >
<!ENTITY   rights "All rights reserved" >
<!ENTITY   book   "La Peste: Albert Camus,
&#xA9; 1947 %pub;. &rights;" >
```

then the replacement text for the entity "`book`" is:

```
La Peste: Albert Camus,
© 1947 Éditions Gallimard. &rights;
```

The general-entity reference "`&rights;`" would be expanded should the reference "`&book;`" appear in the document's content or an attribute value.

These simple rules may have complex interactions; for a detailed discussion of a difficult example, see **D Expansion**

of Entity and Character References.

4.6 Predefined Entities

[Definition: Entity and character references can both be used to **escape** the left angle bracket, ampersand, and other delimiters. A set of general entities (`amp`, `lt`, `gt`, `apos`, `quot`) is specified for this purpose. Numeric character references may also be used; they are expanded immediately when recognized and must be treated as character data, so the numeric character references "`<`" and "`&`" may be used to escape `<` and `&` when they occur in character data.]

All XML processors must recognize these entities whether they are declared or not. For interoperability, valid XML documents should declare these entities, like any others, before using them. If the entities `lt` or `amp` are declared, they must be declared as internal entities whose replacement text is a character reference to the respective character (less-than sign or ampersand) being escaped; the double escaping is required for these entities so that references to them produce a well-formed result. If the entities `gt`, `apos`, or `quot` are declared, they must be declared as internal entities whose replacement text is the single character being escaped (or a character reference to that character; the double escaping here is unnecessary but harmless). For example:

```
<!ENTITY lt     "&#38;#60;">
<!ENTITY gt     "&#62;">
<!ENTITY amp    "&#38;#38;">
<!ENTITY apos   "&#39;">
<!ENTITY quot   "&#34;">
```

4.7 Notation Declarations

[Definition: **Notations** identify by name the format of unparsed entities, the format of elements which bear a notation attribute, or the application to which a processing instruction is addressed.]

[Definition: **Notation declarations** provide a name for the notation, for use in entity and attribute-list declarations and in attribute specifications, and an external identifier for the notation which may allow an XML processor or its client application to locate a helper application capable of processing data in the given notation.]

Notation Declarations

[82]	`NotationDecl`	::=	`'<!NOTATION' S Name S (ExternalID | PublicID) S? '>'`	[VC: Unique Notation Name]
[83]	`PublicID`	::=	`'PUBLIC' S PubidLiteral`	

Validity constraint: Unique Notation Name

Only one notation declaration can declare a given Name.

XML processors must provide applications with the name and external identifier(s) of any notation declared and referred to in an attribute value, attribute definition, or entity declaration. They may additionally resolve the external identifier into the system identifier, file name, or other information needed to allow the application to call a processor for data in the notation described. (It is not an error, however, for XML documents to declare and refer to notations for which notation-specific applications are not available on the system where the XML processor or application is running.)

4.8 Document Entity

[Definition: The **document entity** serves as the root of the entity tree and a starting-point for an XML processor.] This specification does not specify how the document entity is to be located by an XML processor; unlike other entities, the document entity has no name and might well appear on a processor input stream without any identification at all.

5 Conformance

5.1 Validating and Non-Validating Processors

Conforming XML processors fall into two classes: validating and non-validating.

Validating and non-validating processors alike must report violations of this specification's well-formedness constraints in the content of the document entity and any other parsed entities that they read.

[Definition: **Validating processors** must, at user option, report violations of the constraints expressed by the declarations in the DTD, and failures to fulfill the validity constraints given in this specification.] To accomplish this, validating XML processors must read and process the entire DTD and all external parsed entities referenced in the document.

Non-validating processors are required to check only the document entity, including the entire internal DTD subset, for well-formedness. [Definition: While they are not required to check the document for validity, they are required to **process** all the declarations they read in the internal DTD subset and in any parameter entity that they read, up to the first reference to a parameter entity that they do *not* read; that is to say, they must use the information in those declarations to normalize attribute values, include the replacement text of internal entities, and supply default attribute values.] Except when `standalone="yes"`, they must not process entity declarations or attribute-list declarations encountered after a reference to a parameter entity that is not read, since the entity may have contained overriding declarations.

5.2 Using XML Processors

The behavior of a validating XML processor is highly predictable; it must read every piece of a document and report all well-formedness and validity violations. Less is required of a non-validating processor; it need not read any part of the document other than the document entity. This has two effects that may be important to users of XML processors:

- Certain well-formedness errors, specifically those that require reading external entities, may not be detected by a non-validating processor. Examples include the constraints entitled Entity Declared, Parsed Entity, and No Recursion, as well as some of the cases described as forbidden in **4.4 XML Processor Treatment of Entities and References**.
- The information passed from the processor to the application may vary, depending on whether the processor reads parameter and external entities. For example, a non-validating processor may not normalize attribute values, include the replacement text of internal entities, or supply default attribute values, where doing so depends on having read declarations in external or parameter entities.

For maximum reliability in interoperating between different XML processors, applications which use non-validating processors should not rely on any behaviors not required of such processors. Applications which require facilities such as the use of default attributes or internal entities which are declared in external entities should use validating XML processors.

6 Notation

The formal grammar of XML is given in this specification using a simple Extended Backus-Naur Form (EBNF) notation. Each rule in the grammar defines one symbol, in the form

```
symbol ::= expression
```

Symbols are written with an initial capital letter if they are the start symbol of a regular language, otherwise with an initial lower case letter. Literal strings are quoted.

Within the expression on the right-hand side of a rule, the following expressions are used to match strings of one or

more characters:

`#xN`

where `N` is a hexadecimal integer, the expression matches the character in ISO/IEC 10646 whose canonical (UCS-4) code value, when interpreted as an unsigned binary number, has the value indicated. The number of leading zeros in the `#xN` form is insignificant; the number of leading zeros in the corresponding code value is governed by the character encoding in use and is not significant for XML.

`[a-zA-Z]`, `[#xN-#xN]`

matches any Char with a value in the range(s) indicated (inclusive).

`[abc]`, `[#xN#xN#xN]`

matches any Char with a value among the characters enumerated. Enumerations and ranges can be mixed in one set of brackets.

`[^a-z]`, `[^#xN-#xN]`

matches any Char with a value *outside* the range indicated.

`[^abc]`, `[^#xN#xN#xN]`

matches any Char with a value not among the characters given. Enumerations and ranges of forbidden values can be mixed in one set of brackets.

`"string"`

matches a literal string matching that given inside the double quotes.

`'string'`

matches a literal string matching that given inside the single quotes.

These symbols may be combined to match more complex patterns as follows, where `A` and `B` represent simple expressions:

(`expression`)

`expression` is treated as a unit and may be combined as described in this list.

`A?`

matches `A` or nothing; optional `A`.

`A B`

matches `A` followed by `B`. This operator has higher precedence than alternation; thus `A B | C D` is identical to `(A B) | (C D)`.

`A | B`

matches `A` or `B` but not both.

`A - B`

matches any string that matches `A` but does not match `B`.

`A+`

matches one or more occurrences of `A`.Concatenation has higher precedence than alternation; thus `A+ | B+` is identical to `(A+) | (B+)`.

`A*`

matches zero or more occurrences of `A`. Concatenation has higher precedence than alternation; thus `A* | B*` is identical to `(A*) | (B*)`.

Other notations used in the productions are:

`/* ... */`

comment.

`[ wfc: ... ]`

well-formedness constraint; this identifies by name a constraint on well-formed documents associated with a production.

`[ vc: ... ]`

validity constraint; this identifies by name a constraint on valid documents associated with a production.

A References

A.1 Normative References

IANA-CHARSETS

(Internet Assigned Numbers Authority) Official Names for Character Sets, ed. Keld Simonsen et al. See ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

IETF RFC 1766

IETF (Internet Engineering Task Force). RFC 1766: Tags for the Identification of Languages, ed. H. Alvestrand. 1995. (See http://www.ietf.org/rfc/rfc1766.txt.)

ISO/IEC 10646

ISO (International Organization for Standardization). ISO/IEC 10646-1993 (E). Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane. [Geneva]: International Organization for Standardization, 1993 (plus amendments AM 1 through AM 7).

ISO/IEC 10646-2000

ISO (International Organization for Standardization). ISO/IEC 10646-1:2000. Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane. [Geneva]: International Organization for Standardization, 2000.

Unicode

The Unicode Consortium. *The Unicode Standard, Version 2.0.* Reading, Mass.: Addison-Wesley Developers Press, 1996.

Unicode3

The Unicode Consortium. *The Unicode Standard, Version 3.0.* Reading, Mass.: Addison-Wesley Developers Press, 2000. ISBN 0-201-61633-5.

A.2 Other References

Aho/Ullman

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Reading: Addison-Wesley, 1986, rpt. corr. 1988.

Berners-Lee et al.

Berners-Lee, T., R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax and Semantics. 1997. (Work in progress; see updates to RFC1738.)

Brüggemann-Klein

Brüggemann-Klein, Anne. Formal Models in Document Processing. Habilitationsschrift. Faculty of Mathematics at the University of Freiburg, 1993. (See ftp://ftp.informatik.uni-freiburg.de/documents/papers/brueggem/habil.ps.)

Brüggemann-Klein and Wood

Brüggemann-Klein, Anne, and Derick Wood. Deterministic Regular Languages. Universität Freiburg, Institut für Informatik, Bericht 38, Oktober 1991. Extended abstract in A. Finkel, M. Jantzen, Hrsg., STACS 1992, S. 173-184. Springer-Verlag, Berlin 1992. Lecture Notes in Computer Science 577. Full version titled

One-Unambiguous Regular Languages in Information and Computation 140 (2): 229-253, February 1998.

Clark

James Clark. Comparison of SGML and XML. See http://www.w3.org/TR/NOTE-sgml-xml-971215.

IANA-LANGCODES

(Internet Assigned Numbers Authority) Registry of Language Tags, ed. Keld Simonsen et al. (See http://www.isi.edu/in-notes/iana/assignments/languages/.)

IETF RFC2141

IETF (Internet Engineering Task Force). *RFC 2141: URN Syntax,* ed. R. Moats. 1997. (See http://www.ietf.org/rfc/rfc2141.txt.)

IETF RFC 2279

IETF (Internet Engineering Task Force). RFC 2279: UTF-8, a transformation format of ISO 10646, ed. F. Yergeau, 1998. (See http://www.ietf.org/rfc/rfc2279.txt.)

IETF RFC 2376

IETF (Internet Engineering Task Force). RFC 2376: XML Media Types. ed. E. Whitehead, M. Murata. 1998. (See http://www.ietf.org/rfc/rfc2376.txt.)

IETF RFC 2396

IETF (Internet Engineering Task Force). RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. T. Berners-Lee, R. Fielding, L. Masinter. 1998. (See http://www.ietf.org/rfc/rfc2396.txt.)

IETF RFC 2732

IETF (Internet Engineering Task Force). RFC 2732: Format for Literal IPv6 Addresses in URL's. R. Hinden, B. Carpenter, L. Masinter. 1999. (See http://www.ietf.org/rfc/rfc2732.txt.)

IETF RFC 2781

IETF (Internet Engineering Task Force). *RFC 2781: UTF-16, an encoding of ISO 10646*, ed. P. Hoffman, F. Yergeau. 2000. (See http://www.ietf.org/rfc/rfc2781.txt.)

ISO 639

(International Organization for Standardization). ISO 639:1988 (E). Code for the representation of names of languages. [Geneva]: International Organization for Standardization, 1988.

ISO 3166

(International Organization for Standardization). ISO 3166-1:1997 (E). Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes [Geneva]: International Organization for Standardization, 1997.

ISO 8879

ISO (International Organization for Standardization). ISO 8879:1986(E). Information processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML). First edition -- 1986-10-15. [Geneva]: International Organization for Standardization, 1986.

ISO/IEC 10744

ISO (International Organization for Standardization). ISO/IEC 10744-1992 (E). Information technology -- Hypermedia/Time-based Structuring Language (HyTime). [Geneva]: International Organization for Standardization, 1992. *Extended Facilities Annexe.* [Geneva]: International Organization for Standardization, 1996.

WEBSGML

ISO (International Organization for Standardization). ISO 8879:1986 TC2. Information technology -- Document Description and Processing Languages. [Geneva]: International Organization for Standardization, 1998. (See http://www.sgmlsource.com/8879rev/n0029.htm.)

XML Names

Tim Bray, Dave Hollander, and Andrew Layman, editors. Namespaces in XML. Textuality, Hewlett-Packard, and Microsoft. World Wide Web Consortium, 1999. (See http://www.w3.org/TR/REC-xml-names/.)

B Character Classes

Following the characteristics defined in the Unicode standard, characters are classed as base characters (among others, these contain the alphabetic characters of the Latin alphabet), ideographic characters, and combining characters (among others, this class contains most diacritics) Digits and extenders are also distinguished.

Characters

```
[84]  Letter        ::=  BaseChar | Ideographic
[85]  BaseChar      ::=  [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6]
                         | [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131]
                         | [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E]
                         | [#x0180-#x01C3] | [#x01CD-#x01F0] | [#x01F4-#x01F5]
                         | [#x01FA-#x0217] | [#x0250-#x02A8] | [#x02BB-#x02C1]
                         | #x0386 | [#x0388-#x038A] | #x038C | [#x038E-#x03A1]
                         | [#x03A3-#x03CE] | [#x03D0-#x03D6] | #x03DA | #x03DC
                         | #x03DE | #x03E0 | [#x03E2-#x03F3] | [#x0401-#x040C]
                         | [#x040E-#x044F] | [#x0451-#x045C] | [#x045E-#x0481]
                         | [#x0490-#x04C4] | [#x04C7-#x04C8] | [#x04CB-#x04CC]
                         | [#x04D0-#x04EB] | [#x04EE-#x04F5] | [#x04F8-#x04F9]
                         | [#x0531-#x0556] | #x0559 | [#x0561-#x0586]
                         | [#x05D0-#x05EA] | [#x05F0-#x05F2] | [#x0621-#x063A]
                         | [#x0641-#x064A] | [#x0671-#x06B7] | [#x06BA-#x06BE]
                         | [#x06C0-#x06CE] | [#x06D0-#x06D3] | #x06D5
                         | [#x06E5-#x06E6] | [#x0905-#x0939] | #x093D
                         | [#x0958-#x0961] | [#x0985-#x098C] | [#x098F-#x0990]
                         | [#x0993-#x09A8] | [#x09AA-#x09B0] | #x09B2
                         | [#x09B6-#x09B9] | [#x09DC-#x09DD] | [#x09DF-#x09E1]
                         | [#x09F0-#x09F1] | [#x0A05-#x0A0A] | [#x0A0F-#x0A10]
                         | [#x0A13-#x0A28] | [#x0A2A-#x0A30] | [#x0A32-#x0A33]
                         | [#x0A35-#x0A36] | [#x0A38-#x0A39] | [#x0A59-#x0A5C]
                         | #x0A5E | [#x0A72-#x0A74] | [#x0A85-#x0A8B] | #x0A8D
                         | [#x0A8F-#x0A91] | [#x0A93-#x0AA8] | [#x0AAA-#x0AB0]
                         | [#x0AB2-#x0AB3] | [#x0AB5-#x0AB9] | #x0ABD | #x0AE0
                         | [#x0B05-#x0B0C] | [#x0B0F-#x0B10] | [#x0B13-#x0B28]
                         | [#x0B2A-#x0B30] | [#x0B32-#x0B33] | [#x0B36-#x0B39]
                         | #x0B3D | [#x0B5C-#x0B5D] | [#x0B5F-#x0B61]
                         | [#x0B85-#x0B8A] | [#x0B8E-#x0B90] | [#x0B92-#x0B95]
                         | [#x0B99-#x0B9A] | #x0B9C | [#x0B9E-#x0B9F]
                         | [#x0BA3-#x0BA4] | [#x0BA8-#x0BAA] | [#x0BAE-#x0BB5]
                         | [#x0BB7-#x0BB9] | [#x0C05-#x0C0C] | [#x0C0E-#x0C10]
                         | [#x0C12-#x0C28] | [#x0C2A-#x0C33] | [#x0C35-#x0C39]
                         | [#x0C60-#x0C61] | [#x0C85-#x0C8C] | [#x0C8E-#x0C90]
                         | [#x0C92-#x0CA8] | [#x0CAA-#x0CB3] | [#x0CB5-#x0CB9]
                         | #x0CDE | [#x0CE0-#x0CE1] | [#x0D05-#x0D0C]
                         | [#x0D0E-#x0D10] | [#x0D12-#x0D28] | [#x0D2A-#x0D39]
                         | [#x0D60-#x0D61] | [#x0E01-#x0E2E] | #x0E30
                         | [#x0E32-#x0E33] | [#x0E40-#x0E45] | [#x0E81-#x0E82]
                         | #x0E84 | [#x0E87-#x0E88] | #x0E8A | #x0E8D
                         | [#x0E94-#x0E97] | [#x0E99-#x0E9F] | [#x0EA1-#x0EA3]
                         | #x0EA5 | #x0EA7 | [#x0EAA-#x0EAB] | [#x0EAD-#x0EAE]
                         | #x0EB0 | [#x0EB2-#x0EB3] | #x0EBD | [#x0EC0-#x0EC4]
                         | [#x0F40-#x0F47] | [#x0F49-#x0F69] | [#x10A0-#x10C5]
                         | [#x10D0-#x10F6] | #x1100 | [#x1102-#x1103]
                         | [#x1105-#x1107] | #x1109 | [#x110B-#x110C]
```

```
                        | [#x110E-#x1112] | #x113C | #x113E | #x1140 | #x114C
                        | #x114E | #x1150 | [#x1154-#x1155] | #x1159
                        | [#x115F-#x1161] | #x1163 | #x1165 | #x1167 | #x1169
                        | [#x116D-#x116E] | [#x1172-#x1173] | #x1175 | #x119E
                        | #x11A8 | #x11AB | [#x11AE-#x11AF] | [#x11B7-#x11B8]
                        | #x11BA | [#x11BC-#x11C2] | #x11EB | #x11F0 | #x11F9
                        | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9] | [#x1F00-#x1F15]
                        | [#x1F18-#x1F1D] | [#x1F20-#x1F45] | [#x1F48-#x1F4D]
                        | [#x1F50-#x1F57] | #x1F59 | #x1F5B | #x1F5D
                        | [#x1F5F-#x1F7D] | [#x1F80-#x1FB4] | [#x1FB6-#x1FBC]
                        | #x1FBE | [#x1FC2-#x1FC4] | [#x1FC6-#x1FCC]
                        | [#x1FD0-#x1FD3] | [#x1FD6-#x1FDB] | [#x1FE0-#x1FEC]
                        | [#x1FF2-#x1FF4] | [#x1FF6-#x1FFC] | #x2126
                        | [#x212A-#x212B] | #x212E | [#x2180-#x2182]
                        | [#x3041-#x3094] | [#x30A1-#x30FA] | [#x3105-#x312C]
                        | [#xAC00-#xD7A3]
[86] Ideographic    ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]
[87] CombiningChar  ::= [#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
                        | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD]
                        | #x05BF | [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652]
                        | #x0670 | [#x06D6-#x06DC] | [#x06DD-#x06DF]
                        | [#x06E0-#x06E4] | [#x06E7-#x06E8] | [#x06EA-#x06ED]
                        | [#x0901-#x0903] | #x093C | [#x093E-#x094C] | #x094D
                        | [#x0951-#x0954] | [#x0962-#x0963] | [#x0981-#x0983]
                        | #x09BC | #x09BE | #x09BF | [#x09C0-#x09C4]
                        | [#x09C7-#x09C8] | [#x09CB-#x09CD] | #x09D7
                        | [#x09E2-#x09E3] | #x0A02 | #x0A3C | #x0A3E | #x0A3F
                        | [#x0A40-#x0A42] | [#x0A47-#x0A48] | [#x0A4B-#x0A4D]
                        | [#x0A70-#x0A71] | [#x0A81-#x0A83] | #x0ABC
                        | [#x0ABE-#x0AC5] | [#x0AC7-#x0AC9] | [#x0ACB-#x0ACD]
                        | [#x0B01-#x0B03] | #x0B3C | [#x0B3E-#x0B43]
                        | [#x0B47-#x0B48] | [#x0B4B-#x0B4D] | [#x0B56-#x0B57]
                        | [#x0B82-#x0B83] | [#x0BBE-#x0BC2] | [#x0BC6-#x0BC8]
                        | [#x0BCA-#x0BCD] | #x0BD7 | [#x0C01-#x0C03]
                        | [#x0C3E-#x0C44] | [#x0C46-#x0C48] | [#x0C4A-#x0C4D]
                        | [#x0C55-#x0C56] | [#x0C82-#x0C83] | [#x0CBE-#x0CC4]
                        | [#x0CC6-#x0CC8] | [#x0CCA-#x0CCD] | [#x0CD5-#x0CD6]
                        | [#x0D02-#x0D03] | [#x0D3E-#x0D43] | [#x0D46-#x0D48]
                        | [#x0D4A-#x0D4D] | #x0D57 | #x0E31 | [#x0E34-#x0E3A]
                        | [#x0E47-#x0E4E] | #x0EB1 | [#x0EB4-#x0EB9]
                        | [#x0EBB-#x0EBC] | [#x0EC8-#x0ECD] | [#x0F18-#x0F19]
                        | #x0F35 | #x0F37 | #x0F39 | #x0F3E | #x0F3F
                        | [#x0F71-#x0F84] | [#x0F86-#x0F8B] | [#x0F90-#x0F95]
                        | #x0F97 | [#x0F99-#x0FAD] | [#x0FB1-#x0FB7] | #x0FB9
                        | [#x20D0-#x20DC] | #x20E1 | [#x302A-#x302F] | #x3099
                        | #x309A
[88] Digit          ::= [#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9]
                        | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F]
                        | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF]
                        | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F]
                        | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29]
[89] Extender       ::= #x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640 | #x0E46
                        | #x0EC6 | #x3005 | [#x3031-#x3035] | [#x309D-#x309E]
                        | [#x30FC-#x30FE]
```

The character classes defined here can be derived from the Unicode 2.0 character database as follows:

- Name start characters must have one of the categories Ll, Lu, Lo, Lt, Nl.
- Name characters other than Name-start characters must have one of the categories Mc, Me, Mn, Lm, or Nd.
- Characters in the compatibility area (i.e. with character code greater than #xF900 and less than #xFFFE) are not allowed in XML names.
- Characters which have a font or compatibility decomposition (i.e. those with a "compatibility formatting tag" in field 5 of the database -- marked by field 5 beginning with a "<") are not allowed.
- The following characters are treated as name-start characters rather than name characters, because the property file classifies them as Alphabetic: [#x02BB-#x02C1], #x0559, #x06E5, #x06E6.
- Characters #x20DD-#x20E0 are excluded (in accordance with Unicode 2.0, section 5.14).
- Character #x00B7 is classified as an extender, because the property list so identifies it.
- Character #x0387 is added as a name character, because #x00B7 is its canonical equivalent.
- Characters ':' and '_' are allowed as name-start characters.
- Characters '-' and '.' are allowed as name characters.

C XML and SGML (Non-Normative)

XML is designed to be a subset of SGML, in that every XML document should also be a conforming SGML document. For a detailed comparison of the additional restrictions that XML places on documents beyond those of SGML, see [Clark].

D Expansion of Entity and Character References (Non-Normative)

This appendix contains some examples illustrating the sequence of entity- and character-reference recognition and expansion, as specified in **4.4 XML Processor Treatment of Entities and References**.

If the DTD contains the declaration

```
<!ENTITY example "<p>An ampersand (&#38;#38;) may be escaped
numerically (&#38;#38;#38;) or with a general entity
(&amp;amp;).</p>" >
```

then the XML processor will recognize the character references when it parses the entity declaration, and resolve them before storing the following string as the value of the entity "`example`":

```
<p>An ampersand (&#38;) may be escaped
numerically (&#38;#38;) or with a general entity
(&amp;amp;).</p>
```

A reference in the document to "`&example;`" will cause the text to be reparsed, at which time the start- and end-tags of the `p` element will be recognized and the three references will be recognized and expanded, resulting in a `p` element with the following content (all data, no delimiters or markup):

```
An ampersand (&) may be escaped
numerically (&#38;) or with a general entity
(&amp;).
```

A more complex example will illustrate the rules and their effects fully. In the following example, the line numbers are solely for reference.

```
1 <?xml version='1.0'?>
2 <!DOCTYPE test [
3 <!ELEMENT test (#PCDATA) >
4 <!ENTITY % xx '&#37;zz;'>
5 <!ENTITY % zz '&#60;!ENTITY tricky "error-prone" >' >
6 %xx;
7 ]>
8 <test>This sample shows a &tricky; method.</test>
```

This produces the following:

- in line 4, the reference to character 37 is expanded immediately, and the parameter entity "`xx`" is stored in the symbol table with the value "`%zz;`". Since the replacement text is not rescanned, the reference to parameter entity "`zz`" is not recognized. (And it would be an error if it were, since "`zz`" is not yet declared.)
- in line 5, the character reference "`<`" is expanded immediately and the parameter entity "`zz`" is stored with the replacement text "`<!ENTITY tricky "error-prone" >`", which is a well-formed entity declaration.
- in line 6, the reference to "`xx`" is recognized, and the replacement text of "`xx`" (namely "`%zz;`") is parsed. The reference to "`zz`" is recognized in its turn, and its replacement text ("`<!ENTITY tricky "error-prone" >`") is parsed. The general entity "`tricky`" has now been declared, with the replacement text "`error-prone`".
- in line 8, the reference to the general entity "`tricky`" is recognized, and it is expanded, so the full content of the `test` element is the self-describing (and ungrammatical) string *This sample shows a error-prone method.*

E Deterministic Content Models (Non-Normative)

As noted in **3.2.1 Element Content**, it is required that content models in element type declarations be deterministic. This requirement is for compatibility with SGML (which calls deterministic content models "unambiguous"); XML processors built using SGML systems may flag non-deterministic content models as errors.

For example, the content model `((b, c) | (b, d))` is non-deterministic, because given an initial `b` the XML processor cannot know which `b` in the model is being matched without looking ahead to see which element follows the `b`. In this case, the two references to `b` can be collapsed into a single reference, making the model read `(b, (c | d))`. An initial `b` now clearly matches only a single name in the content model. The processor doesn't need to look ahead to see what follows; either `c` or `d` would be accepted.

More formally: a finite state automaton may be constructed from the content model using the standard algorithms, e.g. algorithm 3.5 in section 3.9 of Aho, Sethi, and Ullman [Aho/Ullman]. In many such algorithms, a follow set is constructed for each position in the regular expression (i.e., each leaf node in the syntax tree for the regular expression); if any position has a follow set in which more than one following position is labeled with the same element type name, then the content model is in error and may be reported as an error.

Algorithms exist which allow many but not all non-deterministic content models to be reduced automatically to equivalent deterministic models; see Brüggemann-Klein 1991 [Brüggemann-Klein].

F Autodetection of Character Encodings (Non-Normative)

The XML encoding declaration functions as an internal label on each entity, indicating which character encoding is in use. Before an XML processor can read the internal label, however, it apparently has to know what character encoding is in use--which is what the internal label is trying to indicate. In the general case, this is a hopeless situation. It is not entirely hopeless in XML, however, because XML limits the general case in two ways: each

implementation is assumed to support only a finite set of character encodings, and the XML encoding declaration is restricted in position and content in order to make it feasible to autodetect the character encoding in use in each entity in normal cases. Also, in many cases other sources of information are available in addition to the XML data stream itself. Two cases may be distinguished, depending on whether the XML entity is presented to the processor without, or with, any accompanying (external) information. We consider the first case first.

F.1 Detection Without External Encoding Information

Because each XML entity not accompanied by external encoding information and not in UTF-8 or UTF-16 encoding *must* begin with an XML encoding declaration, in which the first characters must be '`<?xml`', any conforming processor can detect, after two to four octets of input, which of the following cases apply. In reading this list, it may help to know that in UCS-4, '<' is "`#x0000003C`" and '?' is "`#x0000003F`", and the Byte Order Mark required of UTF-16 data streams is "`#xFEFF`". The notation ## is used to denote any byte value except that two consecutive ##s cannot be both 00.

With a Byte Order Mark:

`00 00 FE FF`	UCS-4, big-endian machine (1234 order)
`FF FE 00 00`	UCS-4, little-endian machine (4321 order)
`00 00 FF FE`	UCS-4, unusual octet order (2143)
`FE FF 00 00`	UCS-4, unusual octet order (3412)
`FE FF ## ##`	UTF-16, big-endian
`FF FE ## ##`	UTF-16, little-endian
`EF BB BF`	UTF-8

Without a Byte Order Mark:

`00 00 00 3C`
`3C 00 00 00`
`00 00 3C 00`
`00 3C 00 00`
	UCS-4 or other encoding with a 32-bit code unit and ASCII characters encoded as ASCII values, in respectively big-endian (1234), little-endian (4321) and two unusual byte orders (2143 and 3412). The encoding declaration must be read to determine which of UCS-4 or other supported 32-bit encodings applies.

`00 3C 00 3F`	UTF-16BE or big-endian ISO-10646-UCS-2 or other encoding with a 16-bit code unit in big-endian order and ASCII characters encoded as ASCII values (the encoding declaration must be read to determine which)
`3C 00 3F 00`	UTF-16LE or little-endian ISO-10646-UCS-2 or other encoding with a 16-bit code unit in little-endian order and ASCII characters encoded as ASCII values (the encoding declaration must be read to determine which)
`3C 3F 78 6D`	UTF-8, ISO 646, ASCII, some part of ISO 8859, Shift-JIS, EUC, or any other 7-bit, 8-bit, or mixed-width encoding which ensures that the characters of ASCII have their normal positions, width, and values; the actual encoding declaration must be read to detect which of these applies, but since all of these encodings use the same bit patterns for the relevant ASCII characters, the encoding declaration itself may be read reliably
`4C 6F A7 94`	EBCDIC (in some flavor; the full encoding declaration must be read to tell which code page is in use)
Other	UTF-8 without an encoding declaration, or else the data stream is mislabeled (lacking a required encoding declaration), corrupt, fragmentary, or enclosed in a wrapper of some kind

Note:

In cases above which do not require reading the encoding declaration to determine the encoding, section 4.3.3 still requires that the encoding declaration, if present, be read and that the encoding name be checked to match the actual encoding of the entity. Also, it is possible that new character encodings will be invented that will make it necessary to use the encoding declaration to determine the encoding, in cases where this is not required at present.

This level of autodetection is enough to read the XML encoding declaration and parse the character-encoding identifier, which is still necessary to distinguish the individual members of each family of encodings (e.g. to tell UTF-8 from 8859, and the parts of 8859 from each other, or to distinguish the specific EBCDIC code page in use, and so on).

Because the contents of the encoding declaration are restricted to characters from the ASCII repertoire (however encoded), a processor can reliably read the entire encoding declaration as soon as it has detected which family of encodings is in use. Since in practice, all widely used character encodings fall into one of the categories above, the XML encoding declaration allows reasonably reliable in-band labeling of character encodings, even when external sources of information at the operating-system or transport-protocol level are unreliable. Character encodings such as UTF-7 that make overloaded usage of ASCII-valued bytes may fail to be reliably detected.

Once the processor has detected the character encoding in use, it can act appropriately, whether by invoking a separate input routine for each case, or by calling the proper conversion function on each character of input.

Like any self-labeling system, the XML encoding declaration will not work if any software changes the entity's character set or encoding without updating the encoding declaration. Implementors of character-encoding routines should be careful to ensure the accuracy of the internal and external information used to label the entity.

F.2 Priorities in the Presence of External Encoding Information

The second possible case occurs when the XML entity is accompanied by encoding information, as in some file systems and some network protocols. When multiple sources of information are available, their relative priority and the preferred method of handling conflict should be specified as part of the higher-level protocol used to deliver XML. In particular, please refer to [IETF RFC 2376] or its successor, which defines the `text/xml` and `application/xml` MIME types and provides some useful guidance. In the interests of interoperability, however, the following rule is recommended.

- If an XML entity is in a file, the Byte-Order Mark and encoding declaration are used (if present) to determine the character encoding.

G W3C XML Working Group (Non-Normative)

This specification was prepared and approved for publication by the W3C XML Working Group (WG). WG approval of this specification does not necessarily imply that all WG members voted for its approval. The current and former members of the XML WG are:

- Jon Bosak, Sun (*Chair*)
- James Clark (*Technical Lead*)
- Tim Bray, Textuality and Netscape (*XML Co-editor*)
- Jean Paoli, Microsoft (*XML Co-editor*)
- C. M. Sperberg-McQueen, U. of Ill. (*XML Co-editor*)
- Dan Connolly, W3C (*W3C Liaison*)
- Paula Angerstein, Texcel
- Steve DeRose, INSO
- Dave Hollander, HP
- Eliot Kimber, ISOGEN
- Eve Maler, ArborText
- Tom Magliery, NCSA
- Murray Maloney, SoftQuad, Grif SA, Muzmo and Veo Systems
- MURATA Makoto (FAMILY Given), Fuji Xerox Information Systems
- Joel Nava, Adobe
- Conleth O'Connell, Vignette
- Peter Sharpe, SoftQuad

- John Tigue, DataChannel

H W3C XML Core Group (Non-Normative)

The second edition of this specification was prepared by the W3C XML Core Working Group (WG). The members of the WG at the time of publication of this edition were:

- Paula Angerstein, Vignette
- Daniel Austin, Ask Jeeves
- Tim Boland
- Allen Brown, Microsoft
- Dan Connolly, W3C (*Staff Contact*)
- John Cowan, Reuters Limited
- John Evdemon, XMLSolutions Corporation
- Paul Grosso, Arbortext (*Co-Chair*)
- Arnaud Le Hors, IBM (*Co-Chair*)
- Eve Maler, Sun Microsystems (*Second Edition Editor*)
- Jonathan Marsh, Microsoft
- MURATA Makoto (FAMILY Given), IBM
- Mark Needleman, Data Research Associates
- David Orchard, Jamcracker
- Lew Shannon, NCR
- Richard Tobin, University of Edinburgh
- Daniel Veillard, W3C
- Dan Vint, Lexica
- Norman Walsh, Sun Microsystems
- François Yergeau, Alis Technologies (*Errata List Editor*)
- Kongyi Zhou, Oracle

I Production Notes (Non-Normative)

This Second Edition was encoded in the XMLspec DTD (which has documentation available). The HTML versions were produced with a combination of the xmlspec.xsl, diffspec.xsl, and REC-xml-2e.xsl XSLT stylesheets. The PDF version was produced with the html2ps facility and a distiller program.

DocBook

- What Is DocBook?
- SGML
- XML
- XML Schema
- RELAX Schema
- TREX Schema
- Documentation
- Samples
- Tools
- Mailing Lists
- Meetings
- The OASIS T.C.

OASIS

Text Only

Hello, and Welcome!

This is the official DocBook Homepage. DocBook is a DTD (both SGML and XML versions are available) maintained by the DocBook Technical Committee of OASIS. It is particularly well suited to books and papers about computer hardware and software (though it is by no means limited to these applications).

What's New

12 March 2001

Published updated RELAX and TREX Schemas.

Published DocBook V5.0*alpha 1*.

Published MathML 1.0beta4.

23 February 2001

Published minutes from the 23 February 2001 TC meeting.

01 February 2001

DocBook 4.1 becomes an Official OASIS Specification. (The DocBook 4.1 Specification includes both the DocBook V4.1 DTD and the DocBook XML V4.1.2 DTD.)

12 January 2001

Published experimental RELAX and TREX Schemas for DocBook V4.1.2. Updated the XML Schema version.

10 January 2001

Published minutes from the 07 December TC meeting.

Made small updates to the XML Schema version of DocBook and moved it to the OASIS site.

Copyright © 1998, 1999, 2000, 2001 OASIS.

WD-xlink-19980303

XML Linking Language (XLink)

World Wide Web Consortium Working Draft 3-March-1998

This version:

http://www.w3.org/TR/1998/WD-xlink-19980303

Previous version:

http://www.w3.org/TR/WD-xml-link-970731

Latest version:

http://www.w3.org/TR/WD-xlink

Editors:

Eve Maler (ArborText) <elm@arbortext.com>

Steve DeRose (Inso Corp. and Brown University) <sderose@eps.inso.com>

Status of this document

This is a W3C Working Draft for review by W3C members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". A list of current W3C working drafts can be found at http://www.w3.org/TR.

This work is part of the W3C XML Activity (for current status, see http://www.w3.org/MarkUp/XML/Activity). For information about the XPointer language which is expected to be used with XLink, see http://www.w3.org/TR/WD-xptr.

See http://www.w3.org/TR/NOTE-xlink-principles for additional background on the design principles informing XLink.

Abstract

This document specifies constructs that may be inserted into XML resources to describe links between objects. It uses XML syntax to create structures that can describe the simple unidirectional hyperlinks of today's HTML as well as more sophisticated multi-ended and typed links.

XML Linking Language (XLink)

Version 1.0

Table of Contents

1. Introduction
   1.1 Origin and Goals
   1.2 Relationship to Existing Standards
   1.3 Terminology
   1.4 Notation
2. Locator Syntax
3. Link Recognition
4. Linking Elements
   4.1 Information Associated with Links
      4.1.1 Locators
      4.1.2 Link Semantics
      4.1.3 Remote Resource Semantics
      4.1.4 Local Resource Semantics
   4.2 Simple Links
   4.3 Extended Links
5. Extended Link Groups
6. Link Behavior
   6.1 The "Show" Axis
   6.2 The "Actuate" Axis
   6.3 Combinations of the "Show" and "Actuate" Axes
7. Attribute Remapping
8. Conformance

Appendices

A. Unfinished Work
   A.1 Structured Titles
B. References

---

1. Introduction

This document specifies constructs that may be inserted into XML resources to describe links between objects. A link, as the term is used here, is an explicit relationship between two or more data objects or portions of data objects. This specification is concerned with the syntax used to assert link existence and describe link characteristics. Implicit (unasserted) relationships, for example that of one word to the next or that of a word in a text to its entry in an on-line dictionary are obviously important, but outside its

scope.

Links are asserted by elements contained in XML documents. The simplest case is very like an HTML A link, and has these characteristics:

- The link is expressed at one of its ends (similar to the A element in some document)
- Users can only initiate travel from that end to the other
- The link's effect on windows, frames, go-back lists, stylesheets in use, and so on is mainly determined by browsers, not by the link itself. For example, traversal of A links normally replaces the current view, perhaps with a user option to open a new window.
- The link goes to only one destination (although a server may have great freedom in finding or dynamically creating that destination).

While this set of characteristics is already very powerful and obviously has proven itself highly useful and effective, each of these assumptions also limits the range of hypertext functionality. The linking model defined here provides ways to create links that go beyond each of these specific characteristics, thus providing features previously available mostly in dedicated hypermedia systems.

1.1 Origin and Goals

Following is a summary of the design principles governing XLink:

1. XLink shall be straightforwardly usable over the Internet.
2. XLink shall be usable by a wide variety of link usage domains and of classes of linking application software.
3. The XLink expression language shall be XML.
4. The XLink design shall be prepared quickly.
5. The XLink design shall be formal and concise.
6. XLinks shall be human-readable.
7. XLinks may reside outside the documents in which the participating resources reside.
8. XLink shall represent the abstract structure and significance of links.
9. XLink must be feasible to implement.

1.2 Relationship to Existing Standards

Three standards have been especially influential:

- *HTML:* Defines several SGML element types that represent links.
- *HyTime:* Defines inline and out-of-line link structures and some semantic features, including traversal control and presentation of objects.
- *Text Encoding Initiative Guidelines (TEI P3):* Provide structures for creating links, aggregate objects, and link collections.

Many other linking systems have also informed this design, especially Dexter, FRESS, MicroCosm, and InterMedia.

1.3 Terminology

The following basic terms apply in this document.

element tree

A representation of the relevant structure specified by the tags and attributes in an XML document, based on "groves" as defined in the ISO DSSSL standard.

inline link

Abstractly, a link which serves as one of its own resources. Concretely, a link where the content of the linking element serves as a participating resource. HTML `A`, HyTime `clink`, and TEI `XREF` are all examples of inline links.

link

An explicit relationship between two or more data objects or portions of data objects.

linking element

An element that asserts the existence and describes the characteristics of a link.

local resource

The content of an inline linking element. Note that the content of the linking element could be explicitly pointed to by means of a regular locator in the same linking element, in which case the resource is considered remote, not local.

locator

Data, provided as part of a link, which identifies a resource.

multidirectional link

A link whose traversal can be initiated from more than one of its participating resources. Note that being able to "go back" after following a one-directional link does not make the link multidirectional.

out-of-line link

A link whose content does not serve as one of the link's participating resources . Such links presuppose a notion like extended link groups, which indicate to application software where to look for links. Out-of-line links are generally required for supporting multidirectional traversal and for allowing read-only resources to have outgoing links.

participating resource

A resource that belongs to a link. All resources are potential contributors to a link; participating resources are the actual contributors to a particular link.

remote resource

Any participating resource of a link that is pointed to with a locator.

resource

In the abstract sense, an addressable service or unit of information that participates in a link. Examples include files, images, documents, programs, and query results. Concretely, anything reachable by the use of a locator in some linking element. Note that this term and its definition are taken from the basic specifications governing the World Wide Web.

sub-resource

A portion of a resource, pointed to as the precise destination of a link. As one example, a link might specify that an entire document be retrieved and displayed, but that some specific part(s) of

it is the specific linked data, to be treated in an application-appropriate manner such as indication by highlighting, scrolling, etc.

traversal

The action of using a link; that is, of accessing a resource. Traversal may be initiated by a user action (for example, clicking on the displayed content of a linking element) or occur under program control.

1.4 Notation

The formal grammar for locators is given using a simple Extended Backus-Naur Form (EBNF) location, as described in the XML specification.

2. Locator Syntax

The locator for a resource is typically provided by means of a Uniform Resource Identifier, or URI. XPointers can be used in conjunction with the URI structure, as fragment identifiers or queries, to specify a more precise sub-resource. XPointers can be used in conjunction with URIs to specify a more precise sub-resource.

A locator generally contains a URI, as described in IETF RFCs [IETF RFC 1738] and [IETF RFC 1808]. As these RFCs state, the URI may include a trailing *query* (marked by a leading "?"), and be followed by a "#" and a *fragment identifier*, with the query interpreted by the host providing the indicated resource, and the interpretation of the fragment identifier dependent on the data type of the indicated resource.

In order to locate XML documents and portions of documents, a locator value may contain either a URI or a fragment identifier, or both. Any fragment identifier for pointing into XML must be an XPointer.

Special syntax may be used to request the use of particular processing models in accessing the locator's resource. This is designed to reflect the realities of network operation, where it may or may not be desirable to exercise fine control over the distribution of work between local and remote processors.

Locator

```
[1]   Locator ::= URI
                | Connector ( XPointer | Name)
                | URI Connector (XPointer | Name)
[2] Connector ::= '#' | '|'
[3]       URI ::= URIchar*
```

In this discussion, the term **designated resource** refers to the resource which an entire locator serves to locate. The following rules apply:

- The URI, if provided, locates a resource called the **containing resource** .
- If the URI is not provided, the containing resource is considered to be the document in which the linking element is contained.
- If an XPointer is provided, the designated resource is a **sub-resource** of the containing resource; otherwise the designated resource is the containing resource.

- If the `Connector` is followed directly by a `Name`, the `Name` is shorthand for the XPointer "`id(Name)`"; that is, the sub-resource is the element in the containing resource that has an XML ID attribute whose value matches the `Name`. This shorthand is to encourage use of the robust `id` addressing mode.
- If the connector is "#", this signals an intent that the containing resource is to be fetched as a whole from the host that provides it, and that the XPointer processing to extract the sub-resource is to be performed on the client, that is to say on the same system where the linking element is recognized and processed.
- If the connector is "`|`", no intent is signaled as to what processing model is to be used for accessing the designated resource.

Note that by definition, a URI includes an optional query component.

In the case where the URI contains a query (to be interpreted by the server), information providers and authors of server software are urged to use queries as follows:

Query
[4] `Query ::= 'XML-XPTR=' ( XPointer | Name)`

3. Link Recognition

The existence of a link is asserted by a linking element. Linking elements must be recognized reliably by application software in order to provide appropriate display and behavior. There are several ways link recognition could be accomplished: for example, reserving element type names, reserving attributes, or leaving the matter of recognition entirely up to stylesheets and application software. Reserving attributes provides a balance between giving users control of their own markup language design and keeping the important structural fact "is a link" explicit within documents. Therefore, XLink linking-related elements are recognized based on the use of a designated attribute named `xml:link`. Possible values are `simple` and `extended` (which identify linking elements), as well as `locator`, `group`, and `document` (which identify other related types of elements). An element in whose start-tag such an attribute appears is to be treated as an element of the indicated XLink type as dictated by this specification. For example:

```
<A xml:link="simple" href="http://www.w3.org/">The W3C</A>
```

Note: Subject to definitions to be developed in related standards, the methods described in "7. Attribute Remapping" may be used to rename the reserved attribute.

There are two mechanisms that may be used to associate the `xml:link` and `xml:attributes` attributes with a linking element. The simplest is to provide the attribute explicitly in a start-tag. A less verbose method is to use XML's facilities for declaring default attribute values. For example, the following attribute-list declaration would indicate that all instances of the A element in the current document are XLink simple links:

```
<!ATTLIST A xml:link CDATA #FIXED "simple">
```

4. Linking Elements

XLink defines two types of linking element:

- A simple link, which is usually inline and always one-directional
- A much more general extended link, which may be either inline or out-of-line and must be used for multidirectional links, links originating from read-only resources, and so on.

Both kinds of links can have various types of information associated with them.

4.1 Information Associated with Links

The following information can be associated with a link and its resources:

- One or more locators to identify the remote resources participating in the link; a locator is required for each remote resource
- Semantics of the link
- Semantics of the remote resources
- Semantics of the local resource , if the link is inline

This information is supplied in the form of attributes on linking elements. In the following sections, parameter entities are used to group these attributes.

4.1.1 Locators

A locator string identifies a participating resource. A link must supply a locator for each remote resource.

A locator takes the form of an attribute called `href`. Following is a sample declaration of this attribute, enclosed in a `locator.att` parameter entity.

```
<!ENTITY % locator.att
  "href          CDATA              #REQUIRED"
>
```

4.1.2 Link Semantics

The following semantic information can be provided for a link:

- Whether the link is inline

  If the link is inline, its content counts as a local resource of the link. (However, any locator subelements inside the linking element are *not* considered part of the local resource; they are simply part of the linking element machinery.) If the link is out-of-line, its content does not count as a local resource. Every link is either inline or out-of-line. The inline status of a link is indicated with an attribute called `inline`. It can have the value `true` (the default) or `false`.
- The **role** of the link, to identify to application software the meaning of the link

  Links express various kinds of conceptual relationships between the data objects or portions they connect, in terms of significance to the author and user. Some links may be criticisms, others add support or background, while still others might provide access to demographic information about

a data object (its author's name, version number, etc), or to navigational tools such as index, glossary, and summary. To indicate the part that a link plays in representing information, a link author can optionally provide a string identifying the link's role. The role is indicated with an attribute called `role`. (Note that each resource participating in a link may also be given its own role, as described in "4.1.3 Remote Resource Semantics".)

Following are sample declarations of these attributes, enclosed in a `link-semantics.att` parameter entity.

```
<!ENTITY % link-semantics.att
  "inline        (true|false)        'true'
   role          CDATA               #IMPLIED"
>
```

Because simple links have an attribute called `role` that has a different function, they cannot have a `role` attribute for link semantics. Following is a `simple-link-semantics.att` parameter entity declaration for use in simple linking elements.

```
<!ENTITY % simple-link-semantics.att
  "inline        (true|false)        'true'"
>
```

4.1.3 Remote Resource Semantics

The following semantic information can be provided for the remote resources of a link:

- The role of the resource, to identify to application software the part it plays in the link

  (Note that a link as a whole may also be given its own role, as described in "4.1.2 Link Semantics".) A link author can optionally provide role information in an attribute called `role`.
- A title for the resource, to serve as a displayable caption that explains to users the part the resource plays in the link

  A link author can optionally provide title information in an attribute called `title`. XLink does not require that application software make any particular use of title information.
- Behavior policies to use in traversing to this resource

  A link author can optionally use attributes called `show` and `actuate` to communicate general policies concerning the traversal behavior of the link. The `show` attribute can have one of the values `new`, `replace`, and `embed`; the `actuate` attribute can have one of the values `auto` and `user`. A link author can also optionally use an attribute called `behavior` to communicate detailed instructions for traversal behavior. The contents, format, and meaning of this attribute are unconstrained. (See "6. Link Behavior" for more information on the behavior-related attributes.)

Following are sample declarations of these attributes, enclosed in a `remote-resource-semantics.att` parameter entity.

```
<!ENTITY % remote-resource-semantics.att
  "role          CDATA                #IMPLIED
   title         CDATA                #IMPLIED
   show          (embed|replace|new)  #IMPLIED
   actuate       (auto|user)          #IMPLIED
   behavior      CDATA                #IMPLIED"
>
```

4.1.4 Local Resource Semantics

The following semantic information can be provided for the local resource of a link, if the link is inline:

- The role of the resource, to identify to application software the part it plays in the link

  (Note that a link as a whole may also be given its own role, as described in "4.1.2 Link Semantics".) A link author can optionally provide role information in an attribute called `content-role`.
- A title for the resource, to serve as a displayable caption that explains to users the part the resource plays in the link

  A link author can optionally provide title information in an attribute called `content-title`. XLink does not require that application software make any particular use of title information.

Following are sample declarations of these attributes, enclosed in a `local-resource-semantics.att` parameter entity.

```
<!ENTITY % local-resource-semantics.att
  "content-role  CDATA                #IMPLIED
   content-title CDATA                #IMPLIED"
>
```

4.2 Simple Links

Simple links can be used for purposes that approximate the functionality of a basic HTML A link, but they can also support a limited amount of additional functionality. Simple links have only one locator and thus, for convenience, combine the functions of a linking element and a locator into a single element. As a result of this combination, the simple linking element offers both a locator attribute and all the link and resource semantic attributes.

Following is a sample declaration for a simple link, showing all the possible XLink-related attributes it may have (using the parameter entities provided in "4.1 Information Associated with Links"). The `xml:link` attribute value for a simple link must be `simple`.

```
<!ELEMENT simple ANY>
<!ATTLIST simple
    xml:link      CDATA               #FIXED "simple"
    %locator.att;
    %remote-resource-semantics.att;
    %local-resource-semantics.att;
    %simple-link-semantics.att;
>
```

There are no constraints on the contents of a simple linking element. In the sample declaration above, it is given a content model of ANY to indicate that any content model or declared content is acceptable. In a valid document, every element that is significant to XLink must still conform to the constraints expressed in its governing DTD.

Following is an example of a simple link:

```
<mylink xml:link="simple" title="Citation"
   href="http://www.xyz.com/xml/foo.xml" show="new"
   content-role="Reference">as discussed in Smith(1997)</mylink>
```

This example `mylink` element might have the following element and attribute-list declarations:

```
<!ELEMENT mylink (#PCDATA)>
<!ATTLIST mylink
    xml:link      CDATA               #FIXED "simple"
    href          CDATA               #REQUIRED
    content-role  CDATA               #IMPLIED
>
```

Note that it is meaningful to have an out-of-line simple link, although such links are uncommon. They are called "one-ended" and are typically used to associate discrete semantic properties with locations. The properties might be expressed by attributes on the link, the link's element type name, or in some other way, and are not considered full-fledged resources of the link. Most out-of-line links are extended links, as these have a far wider range of uses.

4.3 Extended Links

An **extended link** differs from a simple link in that it can connect any number of resources, not just one local resource (optionally) and one remote resource, and in that extended links are more often out-of-line than simple links.

The additional capabilities of extended links are required for:

- Enabling outgoing links in documents that cannot be modified to add an inline link
- Creating links to and from resources in formats with no native support for embedded links (such as most multimedia formats)
- Applying and filtering sets of relevant links on demand
- Enabling other advanced hypermedia capabilities

Application software might provide traversal among all of a link's participating resources (subject to semantic constraints outside the scope of this specification) and might signal the fact that a given resource or sub-resource participates in one or more links when it is displayed (even though there is no markup at exactly that point to signal it).

A linking element for an extended link contains a series of child elements that serve as locators. Because an extended link can have more than one remote resource, it separates out linking itself from the mechanisms used to locate each resource (whereas a simple link combines the two).

The linking element itself retains those attributes relevant to the link as a whole and to its local resource, if any. Following is a sample declaration for an extended link (using the parameter entities provided in "4.1 Information Associated with Links"). The `xml:link` attribute value for an extended link must be `extended`.

```
<!ELEMENT extended ANY>
<!ATTLIST extended
    xml:link      CDATA               #FIXED "extended"
    %link-semantics.att;
    %local-resource-semantics.att;
>
```

Attributes relevant to remote resources are expressed on the corresponding contained locator elements. Each remote resource can have its own semantics in relation to the link as a whole. Following is a sample declaration for a locator element, showing all the possible XLink-related attributes it may have (using the parameter entities provided in "4.1 Information Associated with Links"). The `xml:link` attribute value for a locator element must be `locator`.

```
<!ELEMENT locator ANY>
<!ATTLIST locator
    xml:link      CDATA               #FIXED "locator"
    %locator.att;
    %remote-resource-semantics.att;
>
```

Following is an example of an out-of-line extended link:

```
<commentary xml:link="extended" inline="false">
   <locator href="smith2.1" role="Essay"/>
   <locator href="jones1.4" role="Rebuttal"/>
   <locator href="robin3.2" role="Comparison"/>
</commentary>
```

For convenience, defaults for the semantic attributes on locator elements can be specified on the linking element that contains them. If any such attribute is omitted from a locator element, the value provided on the containing linking element is to be used. Following is a sample declaration for an extended link (using the parameter entities provided in "4.1 Information Associated with Links") showing all the possible XLink-related attributes it may have, including the remote resource semantic attributes.

```
<!ELEMENT extended ANY>
<!ATTLIST extended
    xml:link      CDATA               #FIXED "extended"
    %link-semantics.att;
    %local-resource-semantics.att;
    %remote-resource-semantics.att;
>
```

The content of a linking element typically consists only of locator elements; however, the declaration as ANY indicates that any other content may be added. (In a valid document, every element that is significant to XLink must still conform to the constraints expressed in its governing DTD.) Only locator elements that are direct children of the linking element define resources linked by that linking element.

A key issue with out-of-line extended links is how linking application software can manage and find them, particularly when they are stored in completely separate documents from those in which their participating resources appear. XLink provides a mechanism for identifying relevant link-containing documents, which is discussed in "5. Extended Link Groups".

5. Extended Link Groups

Hyperlinked documents are often best processed in groups rather than one at a time. If it is desired to highlight resources to advertise that traversal can be initiated, and if at the same time out-of-line links are being used, it may be an absolute requirement to read other documents to find these links and discover where the resources are.

In these cases, an **extended link group** element, a special kind of extended link, may be used to store a list of links to other documents that together constitute an interlinked group. Each such document is identified by means of an **extended link document** element, a special kind of locator element.

Following are sample declarations for extended link group and extended link document elements, showing all the possible XLink-related attributes they may have (using the parameter entities provided in "4.1 Information Associated with Links"). The `xml:link` attribute value for an extended link group element must be `group`, and the value for an extended link document element must be `document`.

```
<!ELEMENT group (document*)>
<!ATTLIST group
    xml:link      CDATA               #FIXED "group"
    steps         CDATA               #IMPLIED
>
<!ELEMENT document EMPTY>
<!ATTLIST document
    xml:link      CDATA               #FIXED "document"
    %locator.att;
>
```

The `steps` attribute may be used by an author to help deal with the situation where an extended link group directs application software to locate another document, which proves to contain an extended link group of its own. There is a potential for infinite regress, and yet there are situations where processing several levels of extended link groups is useful. The `steps` attribute should have a numeric value that

serves as a hint from the author to any link processor as to how many steps of extended link group processing should be undertaken. It does not have any normative effect.

For example, should a group of documents be organized with a single "hub" document containing all the out-of-line links, it might make sense for each non-hub document to contain an extended link group containing only one reference to the hub document. In this case, the best value for `steps` would be `2`.

6. Link Behavior

Link formatting and link behavior are inextricably connected. In general, formatting involves the appearance or treatment of the link prior to any user action, such as choice of font, color, icons, and other devices to show that a link is present. Behavior focuses on what happens when the link is traversed, such as opening, closing, or scrolling windows or panes; displaying the data from various resources in various ways; testing, authenticating, or logging user and context information; or executing various programs.

XLink does not provide mechanisms for controlling link formatting because it is considered to fall into the domain of stylesheets. Link behavior should ideally also be determined by rules based on link types, resource roles, user circumstances, and other factors. However, XLink does provide a few very general behavior mechanisms because they are commonly considered to reflect major or invariant semantics of link types.

The mechanism that XLink provides allows link authors to signal certain intentions as to the timing and effects of traversal. Such intentions can be expressed along two axes, labeled `show` and `actuate`. These are used to express *policies* rather than *mechanisms* ; any link-processing application software is free to devise its own mechanisms, best suited to the user environment and processing mode, to implement the requested policies.

In many cases, much finer control over the details of traversal behavior, of the type that existing hypertext software typically provides, will be desired. Such fine control of link behavior is outside the scope of this specification. However, the `behavior` attribute is provided as a standard place for authors to provide, and in which application software may look, for detailed behavioral instructions.

6.1 The "Show" Axis

The `show` attribute is used to express a policy as to the context in which a resource that is traversed to should be displayed or processed. It may take one of three values:

`embed`

Indicates that upon traversal of the link, the designated resource should be embedded, for the purposes of display or processing, in the body of the resource and at the location where the traversal started.

`replace`

Indicates that upon traversal of the link, the designated resource should, for the purposes of display or processing, replace the resource where the traversal started.

`new`

Indicates that upon traversal of the link, the designated resource should be displayed or processed in a new context, not affecting that of the resource where the traversal started.

6.2 The "Actuate" Axis

The `actuate` attribute is used to express a policy as to when traversal of a link should occur. It may take one of two values:

`auto`

Indicates that the resource in question should be retrieved when any of the other resources of the same link is encountered, and that the display or processing of the initiating resource is not considered complete until this is done. All auto resources are retrieved in the order specified.

`user`

Indicates that the resource should not be presented until there is an explicit external request for traversal.

6.3 Combinations of the "Show" and "Actuate" Axes

Each combination of the `show` and `actuate` attributes is meaningful. Perhaps the least obvious is `show="replace"` combined with `actuate="auto"`; this could be used in "forwarding" type applications, where when one anchor is display, the other(s) are to replace it without user intervention. Since XLink provides only the most general semantics for links, details of presentation, such as a time delay or beep before forwarding, can be specified on a per-application basis using a style language.

7. Attribute Remapping

XLink provides many attributes that can be attached to linking elements to describe various aspects of links, and each has a default name. It may be desired to use existing elements in XML documents as linking elements, but such elements might already have attributes whose names conflict with those described in this document. To avoid collisions, user-chosen attribute names can be mapped to the default names using the `xml:attributes` attribute.

This attribute must contain an even number of white-space-separated names, which are treated as pairs. In each pair, the first name must be one of the default XLink names (`role`, `href`, `title`, `show`, `inline`, `content-role`, `content-title` , `actuate`, `behavior`, `steps`). The second name, when recognized in the document, will be treated as though it were playing the role assigned to the first. For example, consider a DTD with the following declaration:

```
<!ELEMENT TEXT-BOOK ANY>
<!ATTLIST TEXT-BOOK
        title      CDATA                 #IMPLIED
        role       (PRIMARY|SUPPORTING) #IMPLIED
>
```

If it were desired to use this as a simple link, it would be necessary to remap a couple of attributes. This could be accomplished in the internal subset:

```
<!ATTLIST TEXT-BOOK
        xml:link       CDATA            #FIXED "simple"
        xml:attributes CDATA
                       #FIXED "title xl-title role xl-role"
>
```

Then in the document, the following would be recognized as a simple link:

```
<TEXT-BOOK title="Compilers: Principles, Techniques, and Tools"
           role="PRIMARY" xl-title="Primary Textbook for the Course"
           xl-role="ONLINE-PURCHASE"
           href="/cgi/auth-search?q="+Aho+Sethi+Ullman"/>
```

8. Conformance

An element conforms to XLink if:

1. The element has an `xml:link` attribute whose value is one of the attribute values prescribed by this specification, and
2. the element and all of its attributes and content adhere to the syntactic requirements imposed by the chosen `xml:link` attribute value, as prescribed in this specification.

Note that conformance is assessed at the level of individual elements, rather than whole XML documents, because XLink and non-XLink linking mechanisms may be used side by side in any one document.

An application conforms to XLink if it interprets XLink-conforming elements according to all required semantics prescribed by this specification and, for any optional semantics it chooses to support, supports them in the way prescribed.

Appendices

A. Unfinished Work

A.1 Structured Titles

The simple title mechanism described in this draft is insufficient to cope with internationalization or the use of multimedia in link titles. A future version will provide a mechanism for the use of structured link titles.

B. References

XPTR

Eve Maler and Steve DeRose, editors. XML Pointer Language (XPointer) V1.0. ArborText, Inso, and Brown University. Burlington, Seekonk, et al.: World Wide Web Consortium, 1998. (See http://www.w3.org/TR/WD-xptr .)

ISO/IEC 10744

ISO (International Organization for Standardization). ISO/IEC 10744-1992 (E). Information technology --Hypermedia/Time-based Structuring Language (HyTime). [Geneva]: International Organization for Standardization, 1992. Extended Facilities Annex. [Geneva]: International

Organization for Standardization, 1996. (See http://www.ornl.gov/sgml/wg8/hytime/html/is10744r.html).

IETF RFC 1738

IETF (Internet Engineering Task Force). RFC 1738: Uniform Resource Locators. 1991. (See http://www.w3.org/Addressing/rfc1738.txt).

IETF RFC 1808

IETF (Internet Engineering Task Force). RFC 1808: Relative Uniform Resource Locators. 1995. (See http://www.w3.org/Addressing/rfc1808.txt).

TEI

C. M. Sperberg-McQueen and Lou Burnard, editors. Guidelines for Electronic Text Encoding and Interchange. Association for Computers and the Humanities (ACH), Association for Computational Linguistics (ACL), and Association for Literary and Linguistic Computing (ALLC). Chicago, Oxford: Text Encoding Initiative, 1994.

CHUM

Steven J. DeRose and David G. Durand. 1995. "The TEI Hypertext Guidelines." In Computing and the Humanities 29(3). Reprinted in Text Encoding Initiative: Background and Context, ed. Nancy Ide and Jean Véronis, ISBN 0-7923-3704-2.

Copyright © 1998 W3C (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.